COEN-2710 Microprocessors - Lecture 4

Processor Part 1: Pipelining (Ch.4)

Cristinel Ababei
Marquette University
Department of Electrical and Computer Engineering

1

Outline

- Pipelining Introduction
- Pipelined Datapath
- Pipeline Control
- Data Hazards
  - Address data hazards – forwarding
  - Double data hazards - revised forwarding
  - Load-use data hazards - hazard detection unit to “Stall”
- Branch Hazards
  - “Flush”
- Exceptions
  - Exception Handling Routine
- Improving Performance

2

2

Pipelining is Natural

Laundry Example

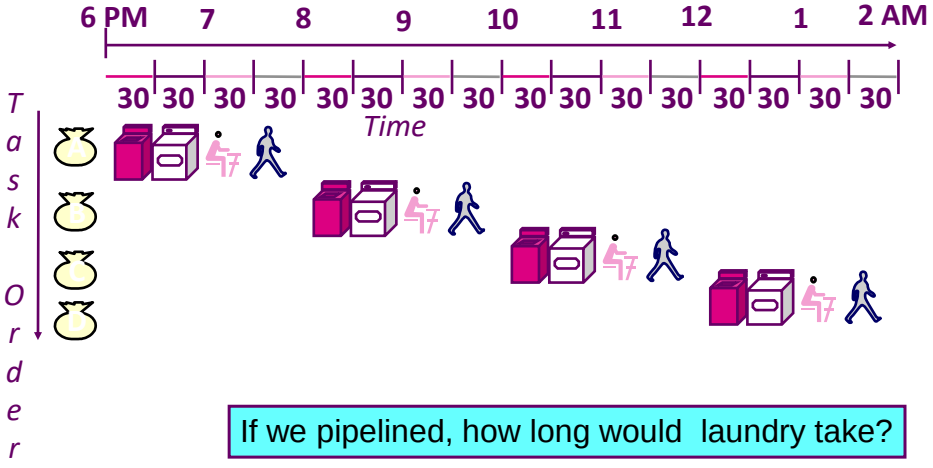
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - "Folder" takes 30 minutes
  - "Stasher" takes 30 minutes to put clothes into drawers

3

3

Sequential Laundry

- Sequential laundry takes 8 hours for 4 loads



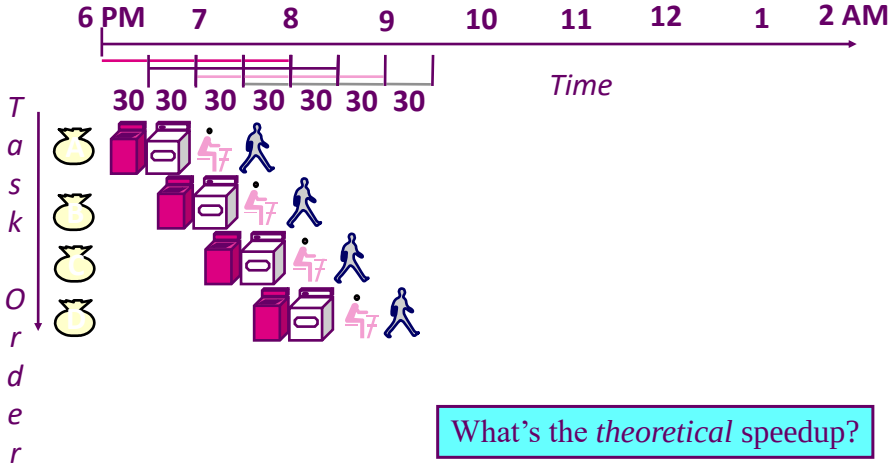
If we pipelined, how long would laundry take?

4

4

Pipelined Laundry: Start work ASAP

- Pipelined laundry takes 3.5 hours for 4 loads!

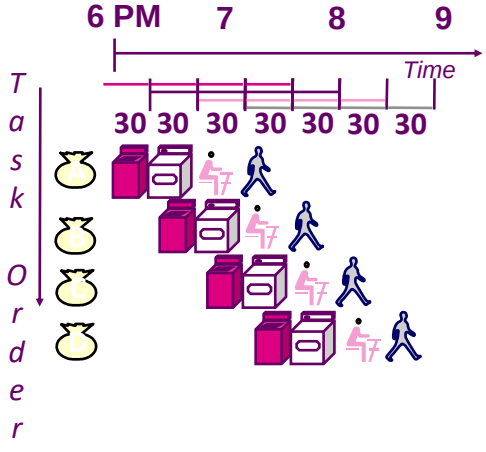


What's the *theoretical* speedup?

5

5

Pipelining Lessons

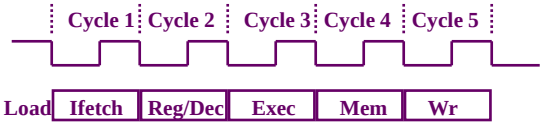


- Pipelining doesn't help latency of a single task, it helps throughput of entire workload
- Max potential speedup is Number of pipe stages
- Performance limited by slowest pipeline stage (Unbalanced lengths of pipe)
- Times to "fill" pipeline and to "drain" it - reduce speedup
- Might need to stall for dependencies

6

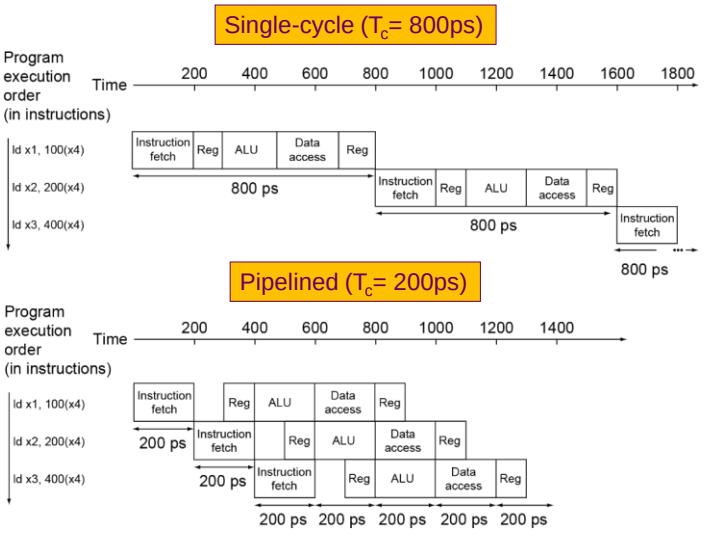
6

The Five Stages of "Load"



1. IF: Get instruction from Instruction Memory
2. ID: Register Fetch and Instruction Decode
3. EX: Calculate the memory address
4. MEM: Read the data from data Memory
5. WB: Write the data back to the register file

Pipeline Performance



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

9

9

Pipelining and ISA Design

- RISC-V ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage

10

10

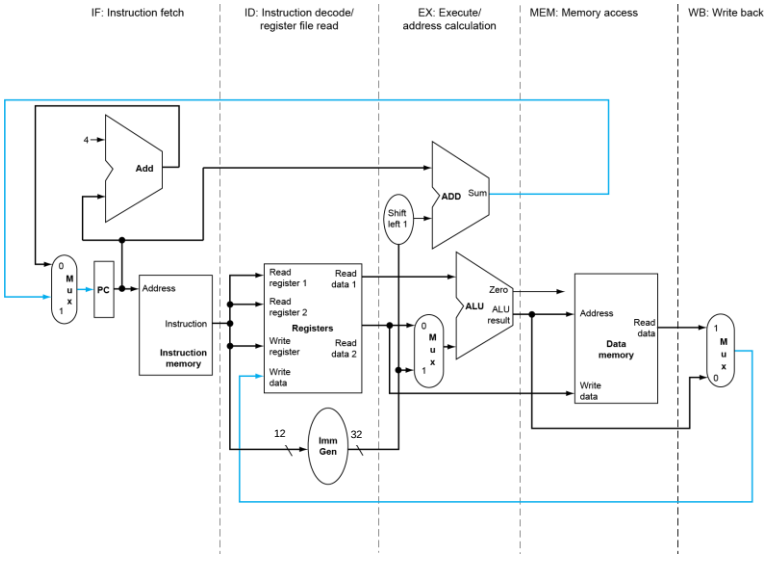
Implementing Pipelining

- What makes it easy
  - Instructions are all the same length
  - Small number of instruction formats
  - Memory operands appear only in loads and stores
- What makes it hard
  - Structural hazards
    - Multiple instructions needing the same datapath components
  - Data hazards
    - Instructions depending on previous instruction's output
  - Control hazards
    - Changes to instruction sequence - branching, jumping
  - Exception handling: overflows, interrupts

11

11

RISC-V Pipelined Datapath

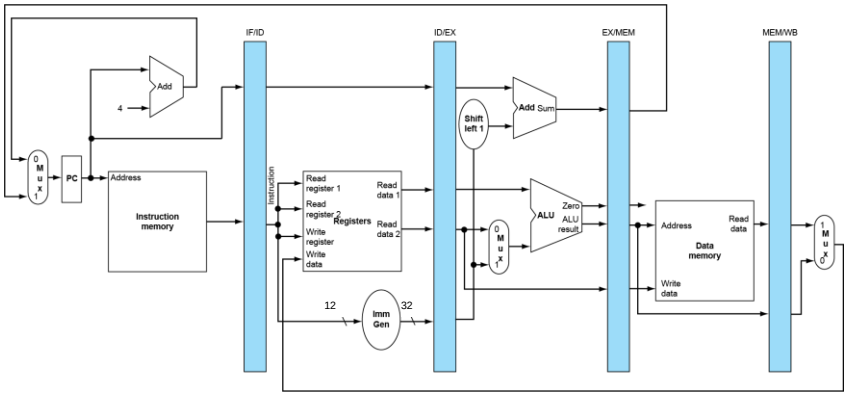


12

12

Pipeline registers

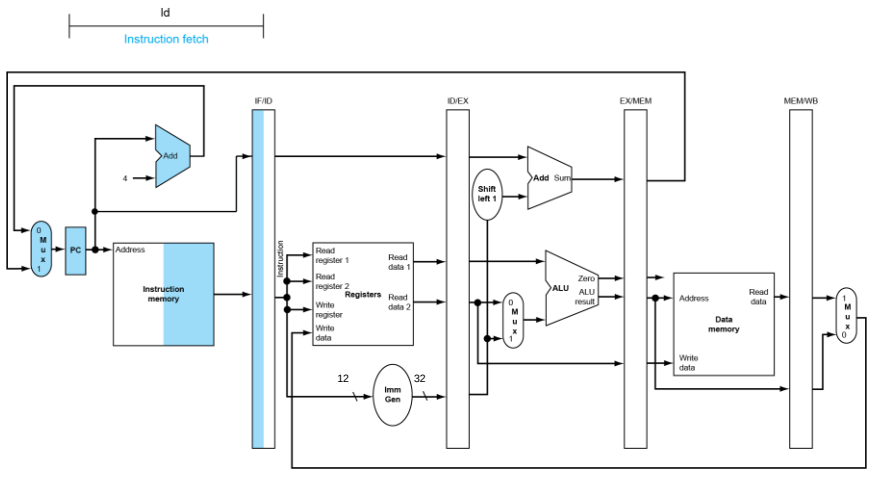
- Need registers between stages
  - To hold information produced in previous cycle



Pipeline Operation

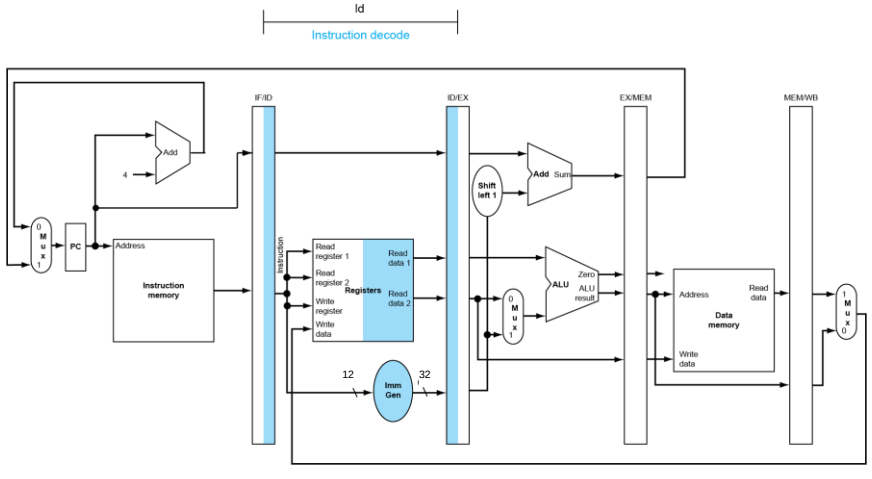
- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for Load & Store

IF for Load, Store, …



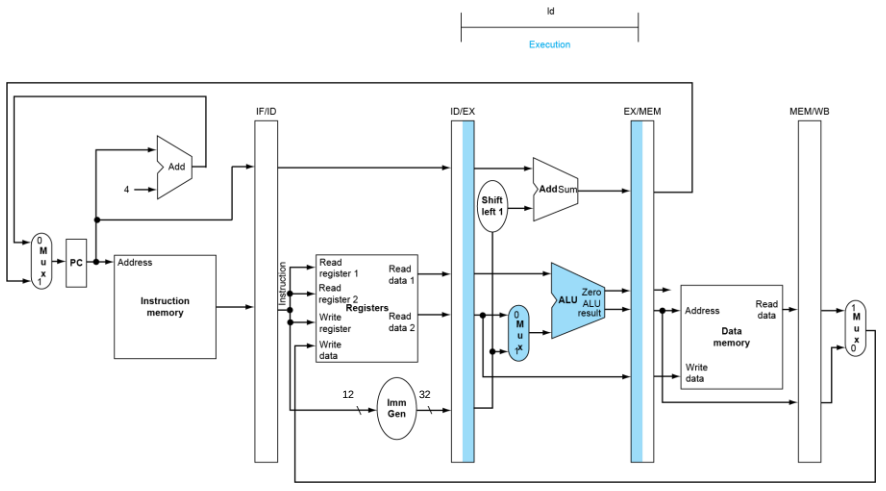
15

ID for Load, Store, …



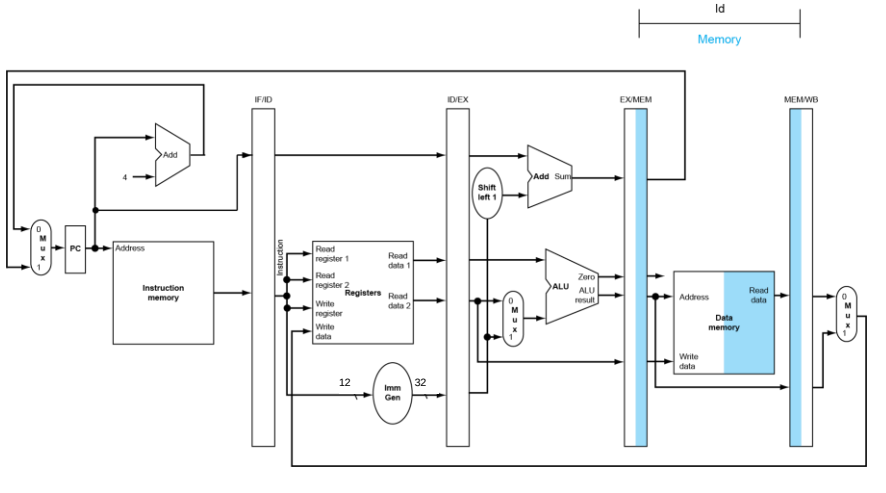
16

EX for Load



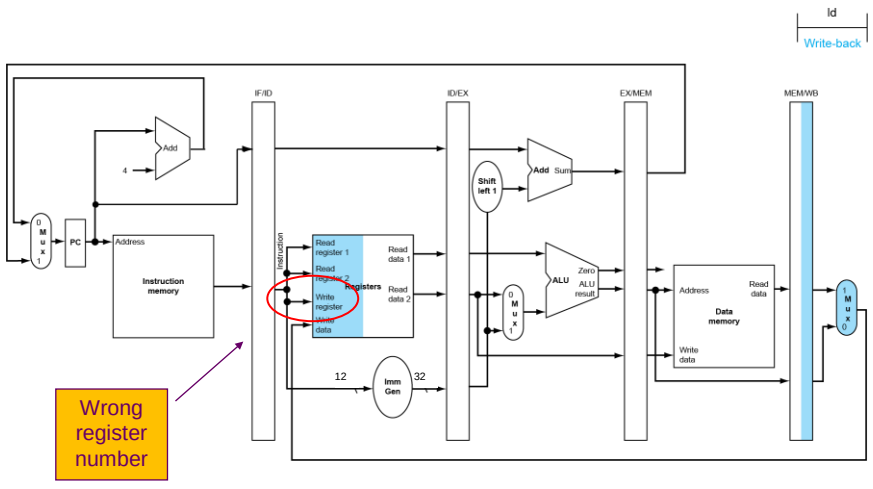
17

MEM for Load

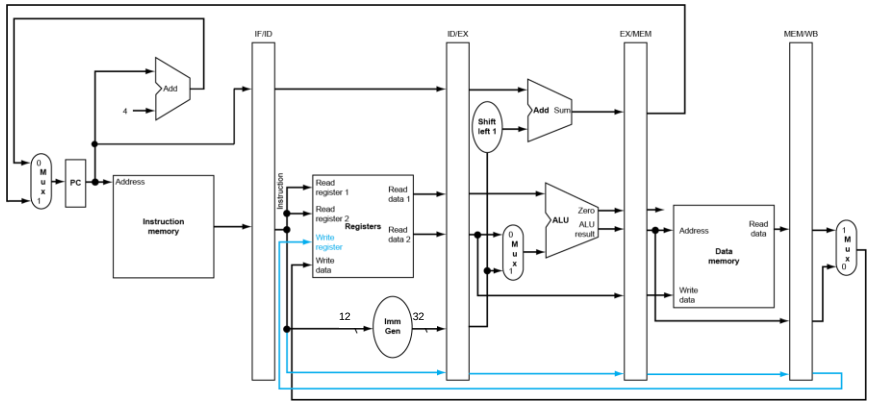


18

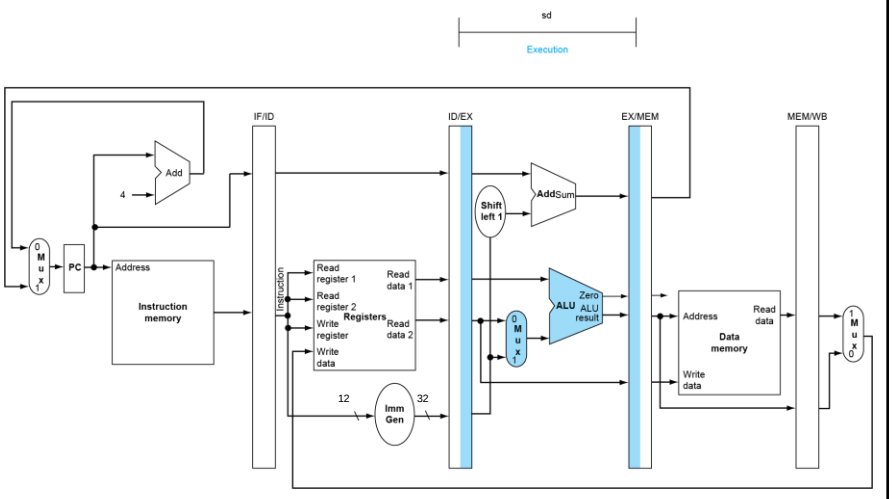
WB for Load



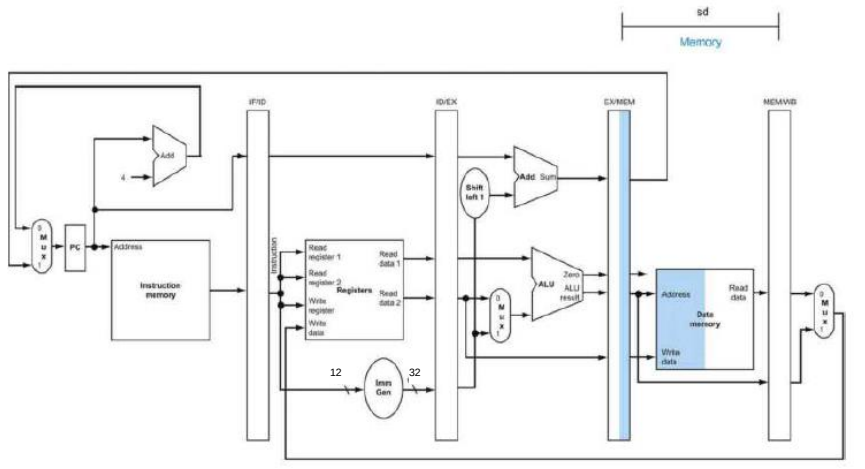
Corrected Datapath for Load



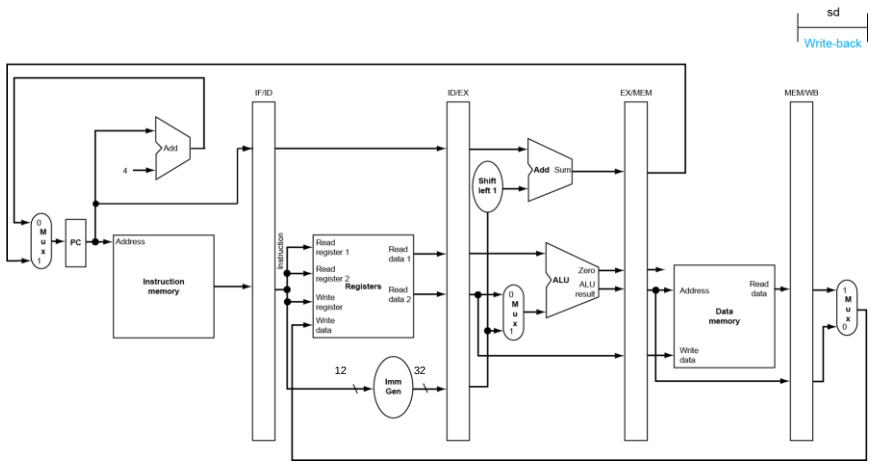
EX for Store



MEM for Store



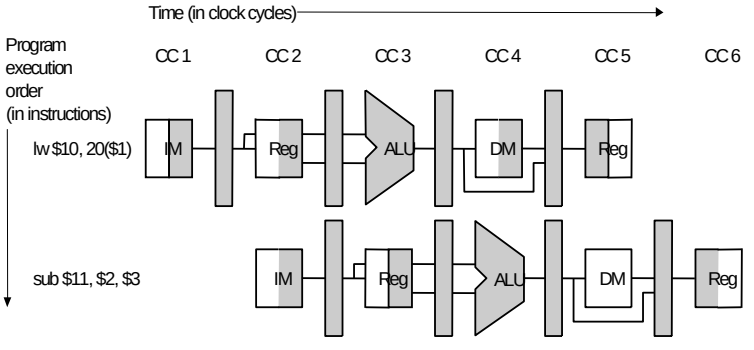
WB for Store



23

23

Graphically Representing Pipelines



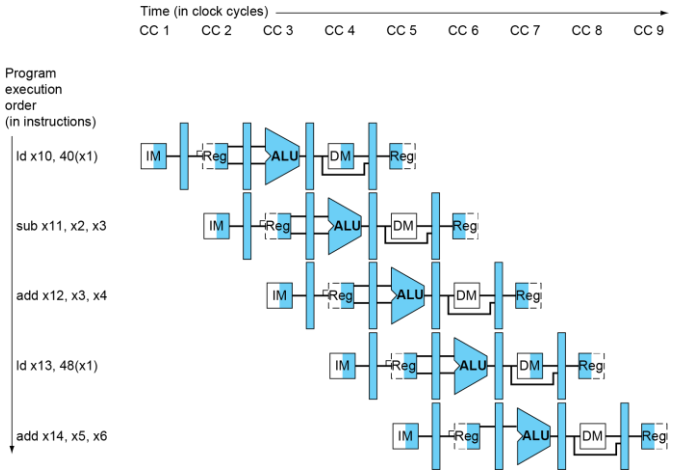
- Can help with answering questions like:
  - How many cycles does it take to execute this code?
  - What is the ALU doing during cycle 4?
- Grayed portions – what's being used for ***this*** instruction.

24

24

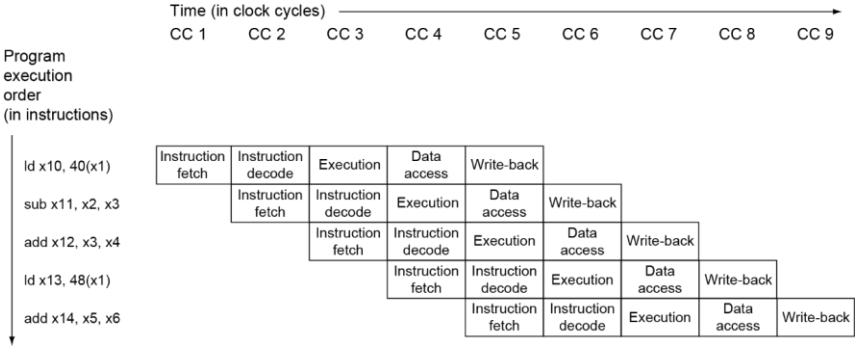
Multi-Cycle Pipeline Diagram

- Form showing resource usage

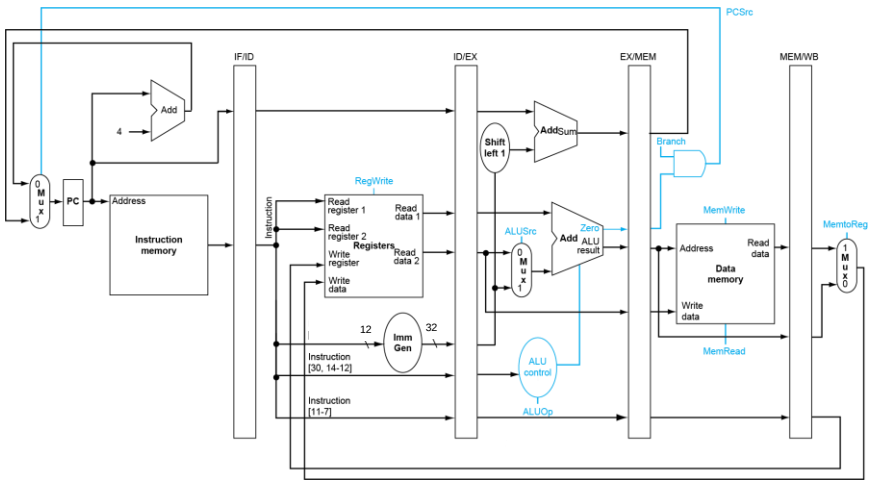


Multi-Cycle Pipeline Diagram

- Traditional form



Pipeline Control (Simplified)



27

27

Pipeline Control

We have 5 separate stages

1. IF: Instruction Fetch and PC Increment
2. ID: Instruction Decode / Register Fetch
3. EX: Execution
4. MEM: Memory Access
5. WR: Write Back

- How should control be handled?
  - A fancy control center telling everyone what to do?
  - Should we use a state machine?

What lines need to be controlled in each stage?

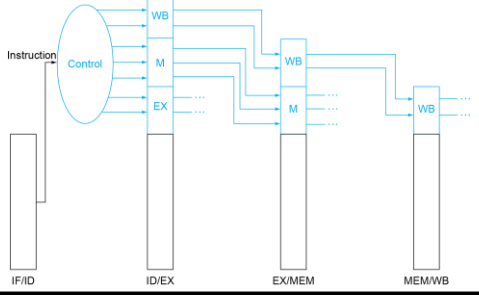
28

28

Pipeline Control

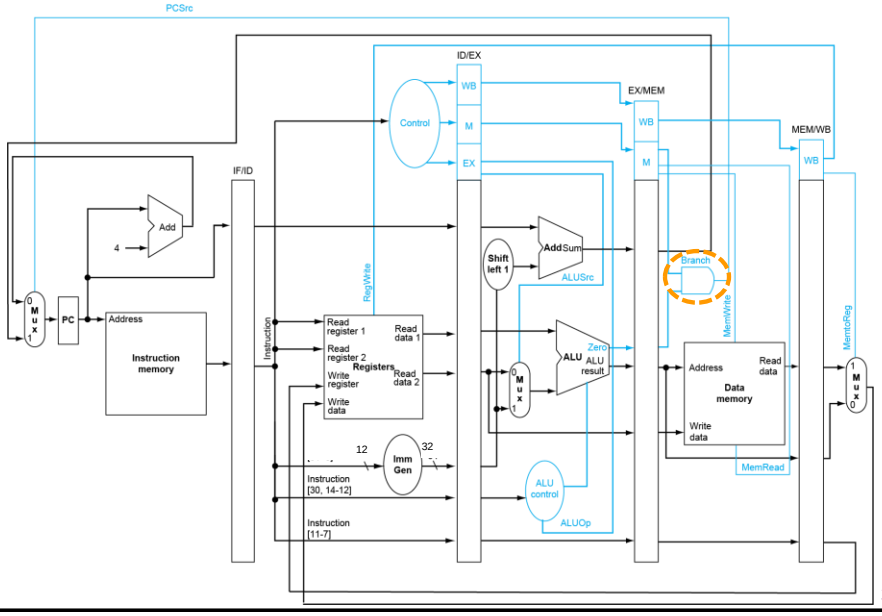
- Control signals derived from instruction
  - As in single-cycle implementation
- Pass control signals along just like the data

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`Load`	0	0	0	1	0	1	0	1	1
`Store`	X	0	0	1	0	0	1	0	X
`BEQ`	X	0	1	0	1	0	0	0	X



29

Datapath with Control



30

Pipeline Summary

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction (in isolation) has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

Outline

- Pipelining Introduction
- Pipelined Datapath
- Pipeline Control
- Data Hazards
  - Address data hazards – forwarding
  - Double data hazards - revised forwarding
  - Load-use data hazards - hazard detection unit to "STALL"
- Branch Hazards
  - "FLUSH"
- Exceptions
  - Exception Handling Routine
- Improving Performance

Hazards

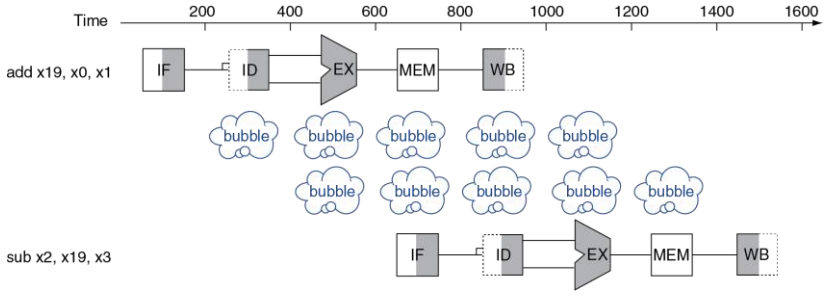
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

1) Structure Hazards

- Conflict for use of a resource
- In RISC-V pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline "bubble"
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

2 - A) Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - `add  x19, x0, x1`
    `sub  x2, x19, x3`



35

35

Data Hazards in ALU Instructions

- Consider this sequence:

```
sub  x2, x1,x3
and  x12,x2,x5
or   x13,x6,x2
add  x14,x2,x2
sd   x15,100(x2)
```

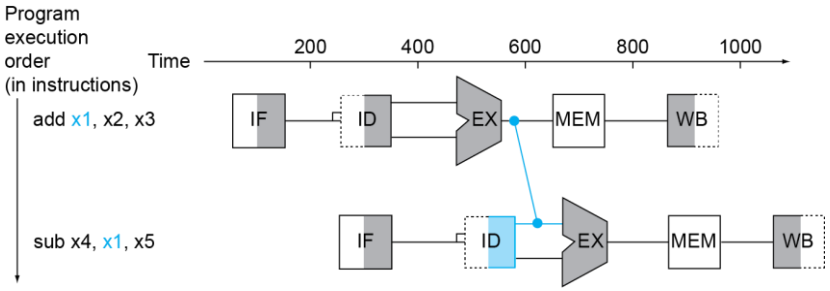
- We can resolve hazards with **forwarding**
  - How do we detect when to forward?

36

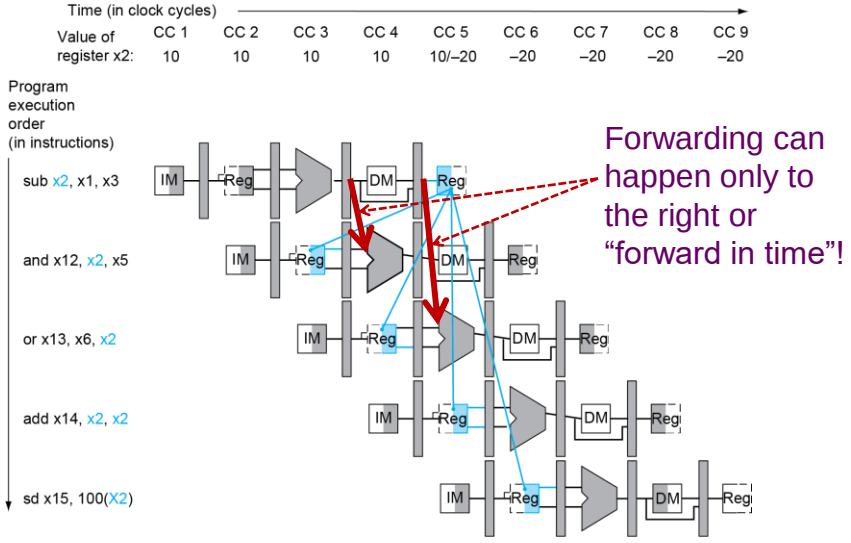
36

Forwarding (aka ByPassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



Dependencies & Forwarding



Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs1 = register number for Rs1 sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs1, ID/EX.RegisterRs2
- Data hazards when
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2

  Fwd from EX/MEM pipeline reg
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

  Fwd from MEM/WB pipeline reg

39

39

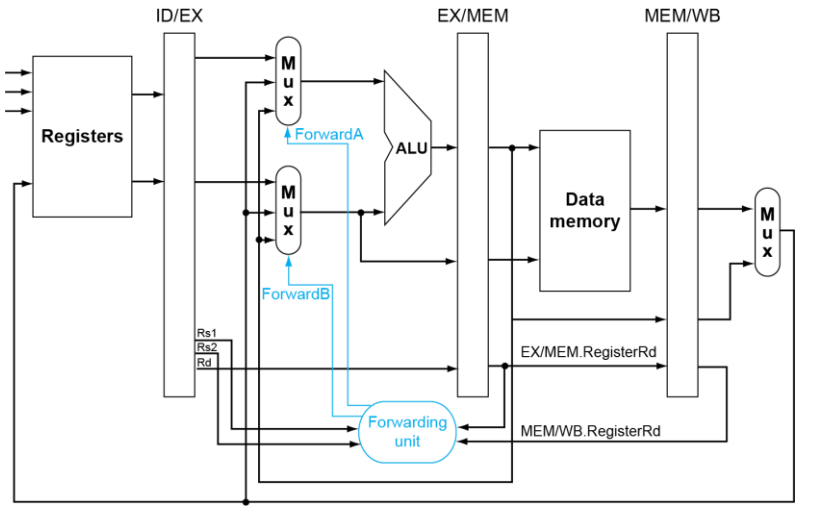
Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not x0
  - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0

40

40

Forwarding Paths



41

41

Forwarding Conditions

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

42

42

Double Data Hazard

- Consider the sequence:

```
add x1,x1,x2
add x1,x1,x3
add x1,x1,x4
```

- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

43

43

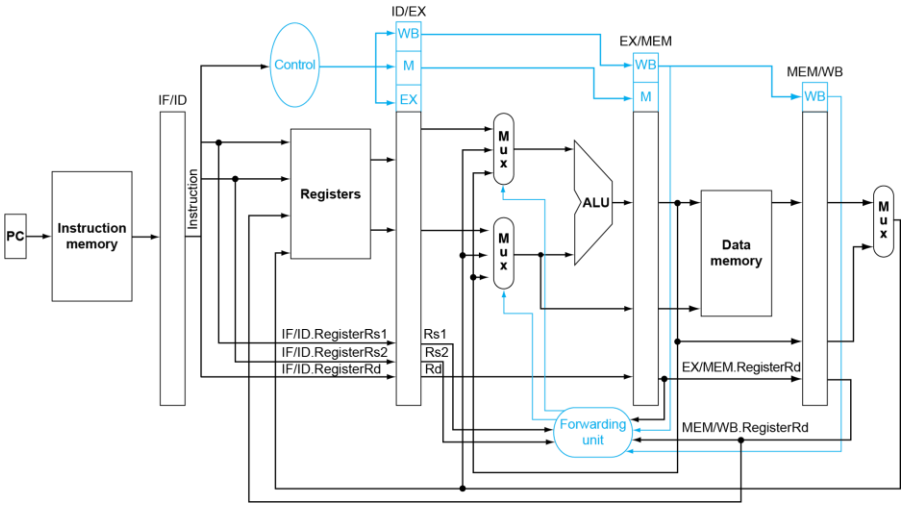
Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 0)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
      and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs1))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 0)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
      and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs2))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01

44

44

Datapath with Forwarding

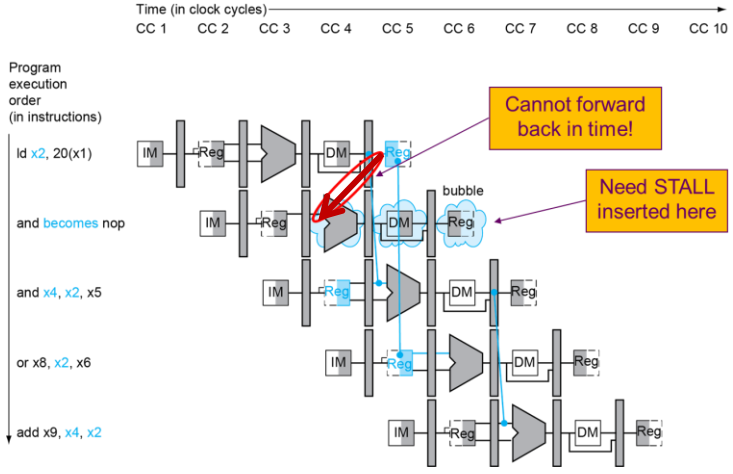


45

45

2 - B) Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



46

46

Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs1, IF/ID.RegisterRs2
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
    (ID/EX.RegisterRd = IF/ID.RegisterRs1))
- If detected, STALL and insert Bubble

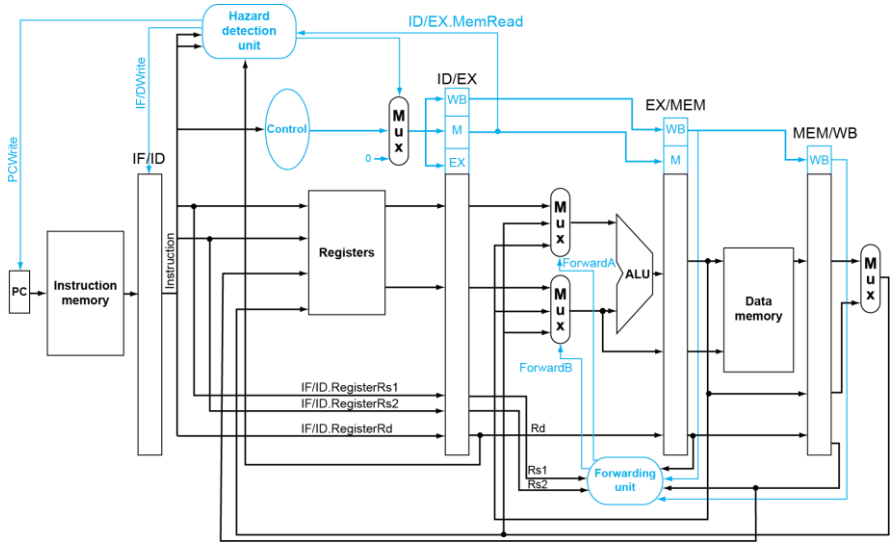
47

How to STALL the Pipeline

1. Force control values in ID/EX register to 0
   - EX, MEM and WB will therefore do NOP (no-operation)
2. Prevent update of PC and IF/ID register
   - Using instruction is decoded again
   - Following instruction is fetched again
   - 1-cycle stall allows MEM to read data for `ld`
     - Can subsequently forward to EX stage

48

Datapath with Hazard Detection



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `a = b + e; c = b + f;`

```
stall   ld   x1, 0(x0)
        ld   x2, 8(x0)
stall   add  x3, x1, x2
        sd   x3, 24(x0)
        ld   x4, 16(x0)
stall   add  x5, x1, x4
        sd   x5, 32(x0)
```

13 cycles

```
ld   x1, 0(x0)
ld   x2, 8(x0)
ld   x4, 16(x0)
add  x3, x1, x2
sd   x3, 24(x0)
add  x5, x1, x4
sd   x5, 32(x0)
```

11 cycles

Stalls and Performance

The BIG Picture

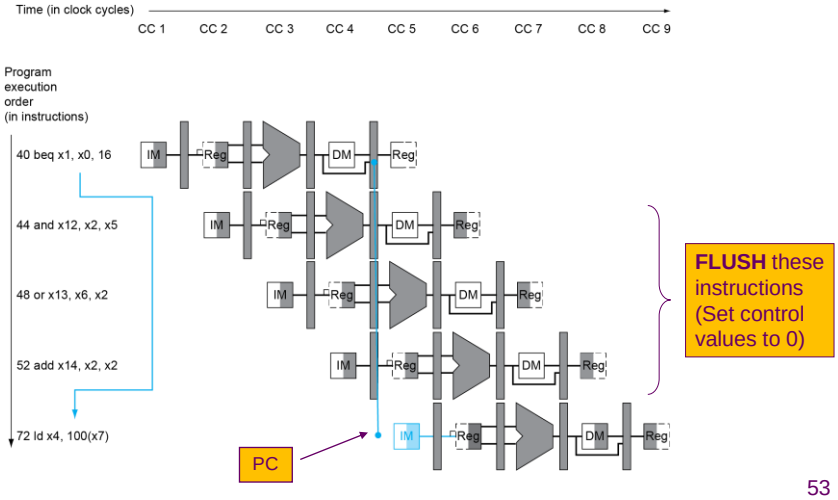
- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

3) Branch (Control) Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- In RISC-V pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Branch Hazards

- If branch outcome determined at the end of EX stage
- By the time we know to branch, new instructions are in the pipe!



Predict!

- Default operation = We are basically predicting "branch not taken"
- What if it is?
  - Toss all instructions in the pipe
  - Keep going (the branch instruction will update the PC correctly and we'll get to the right place)
  - Undo any memory/register changes (this won't happen, since that's at the end of the pipe)

So – how do we tell the datapath to quit executing an instruction in the middle of the pipeline?

FLUSH

- To flush the pipe (when branch is taken)
  - Set IF.Flush (new control line)
  - Zero all control lines
    - Similar to STALL, except don't disable the PC and IF/ID write controls – this effectively writes over what the previous instruction was doing.
  - No memory or register writes will have yet happened, so everything else is OK

Could we make the decision quicker, to lose fewer clock cycles?

55

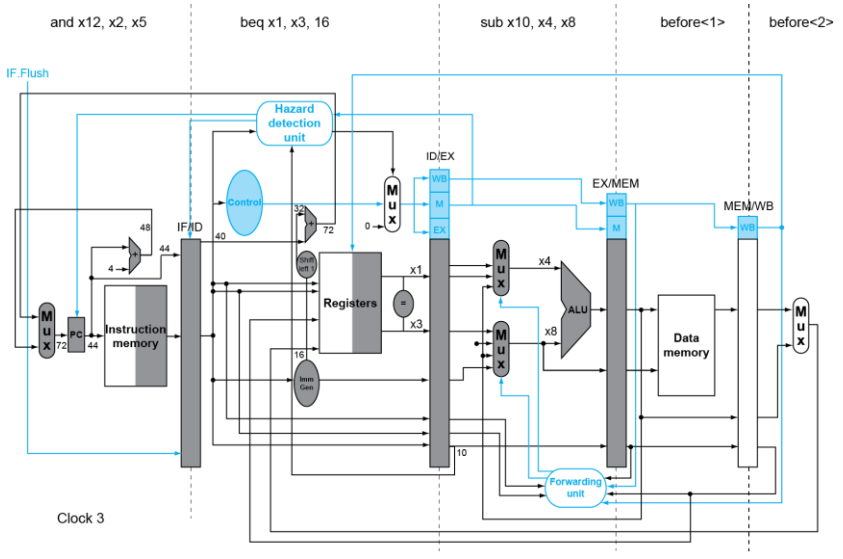
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- This means the branch decision can be made during the ID stage instead of the EX stage.
  - This is why we need an IF.Flush, but *not* an ID.Flush!

But Problem: Messes up forwarding for branching

56

Example: Branch Taken

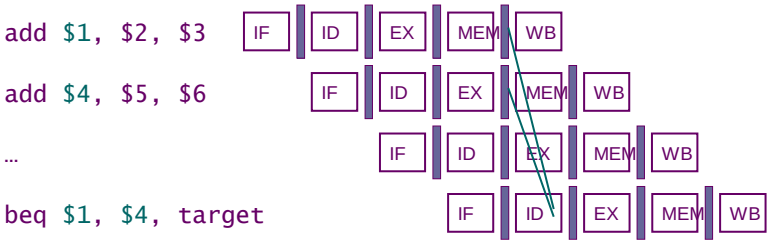


57

57

Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction



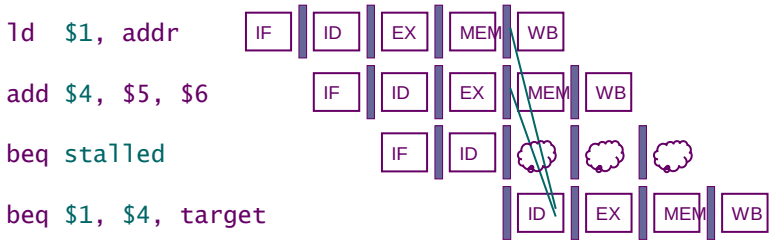
- Can resolve using forwarding

58

58

Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 STALL cycle

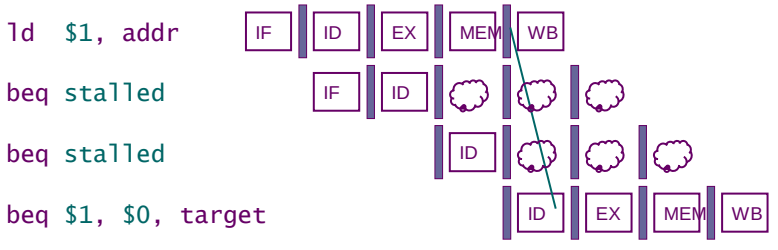


59

59

Data Hazards for Branches

- If a comparison register is a destination of immediately preceding LOAD instruction
  - Need 2 STALL cycles



60

60

Other branch hazard methods

- Dynamic branch prediction
  - Keep a table of bits associated with the current chunk of memory, telling whether a branch was taken last time that instruction was executed. Use that to guess at the next instruction to execute
- Delayed branching
  - Make the instruction set operate such that at a branch the following instruction is always executed, *then* the branch actually occurs. Puts the work on the programmer / assembler. MIPS does this!

61

61

Outline

- Pipelining Introduction
- Pipelined Datapath
- Pipeline Control
- Data Hazards
  - Address data hazards – forwarding
  - Double data hazards - revised forwarding
  - Load-use data hazards - hazard detection unit to “Stall”
- Branch Hazards
  - “Flush”
- Exceptions
  - Exception Handling Routine
- Improving Performance

62

62

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

63

Handling Exceptions

- Save PC of offending (or interrupted) instruction
  - In RISC-V: Supervisor Exception Program Counter (SEPC)
- Save indication of the problem
  - In RISC-V: Supervisor Exception Cause Register (SCAUSE)
  - 64 bits, but most bits unused
    - Exception code field: 2 for undefined opcode, 12 for hardware malfunction, …
- Jump to handler
  - Assume at 0000 0000 1C09 0000_{hex}

64

An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Exception vector address to be added to a vector table base register:
  - Undefined opcode 00 0100 0000_{two}
  - Hardware malfunction: 01 1000 0000_{two}
  - …: …
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

65

65

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - Use SEPC to return to program
- Otherwise
  - Terminate program
  - Report error using SEPC, SCAUSE, …

66

66

Exceptions in a Pipeline

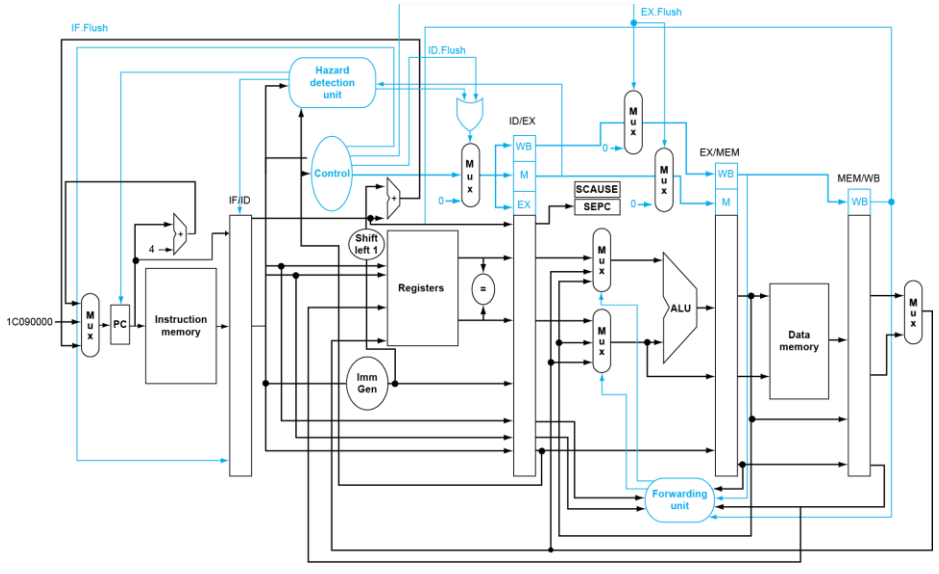
- Another form of control hazard
- Consider malfunction on add in EX stage

  `add x1, x2, x1`
  - Prevent x1 from being clobbered
  - Complete previous instructions
  - Flush `add` and subsequent instructions
  - Set SEPC and SCAUSE register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

67

67

Pipeline with Exceptions



68

68

Improving Performance

- Try to avoid stalls! e.g., reorder these instructions:

```
lw t0, 0(t1)
lw t2, 4(t1)
sw t2, 0(t1)
sw t0, 4(t1)
```

69

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

70

Super-Pipelining

- This is just creating longer pipelines. Remember that the speedup is directly related to the number of stages in the pipe.
- Big issue:
  - How do you break up a datapath into smaller pieces, each with smaller and roughly equal latencies!

71

Superscalar Processing

- Could you do laundry faster if you had 3 washers and 3 dryers?
  - Yes, if you could keep them moving smoothly
- Basic method:
  - Put multiple copies of datapath into hardware
  - Launch multiple instructions every clock cycle
  - Get ready to have serious control and hazard issues!
- Biggest issue: need to schedule instructions quickly and efficiently and get them into the pipes, in such a way that hazards are minimized.

72

Dynamic Pipeline Scheduling

- This is like regular or super pipelining, except with very advanced techniques to make sure there are always instructions ready to fill in if a stall happens (don't want to waste a clock cycle.)
- Basically keeps a series of instruction buffers and schedule the instructions as needed to keep the pipe full and the program running without hazards.
  - Branch prediction / speculative execution
- Can create quite a problem for accurate exception handling.

73

73

Current processors

- Nearly all have pipelining/superscalar design
- Compiler technology important
  - Can't create efficient code for a pipelined or superscalar processor without the compiler thoroughly understanding how the pipelining, scheduling, forwarding and hazard detection all work for that specific processor.

74

74

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction is NOT reduced!
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall