

Computer Architecture

ILP Starters: Inorder Pipelining

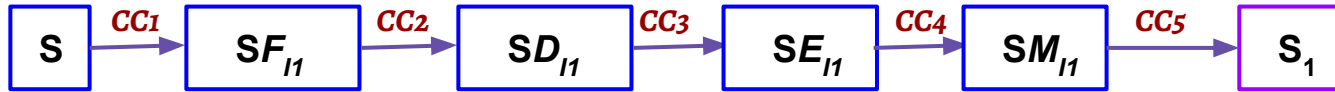
Debadatta Mishra, CSE, IITK

Pipelining overview

- Software visible state transition (**S** $\rightarrow$ **Q**) in **K** clock cycles for a given instruction
- While **I**th instruction is in 2nd step (stage), (**I+1**)th instruction performs step 1
- Non-overlapping use of functional units/storage elements
- Implementation must address the issues due to changes to state by different instructions in the same clock cycle

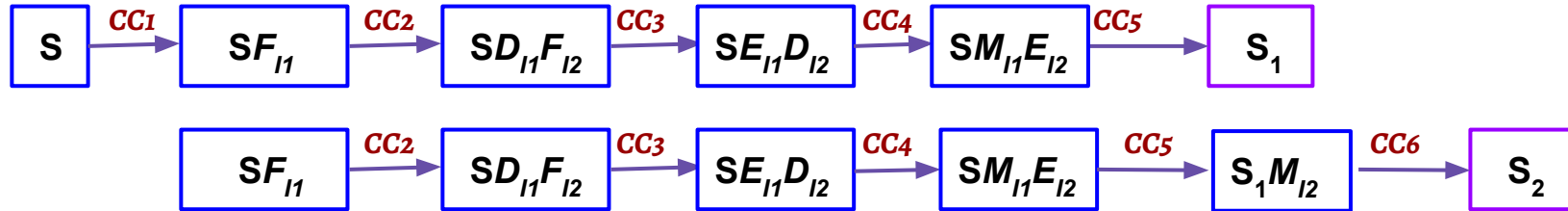
Pipelining overview

- Software visible state transition ($\mathbf{S} \rightarrow \mathbf{Q}$) in $\mathbf{K}$ clock cycles for a given instruction
- While $\mathbf{I}^{\text{th}}$ instruction is in 2nd step (stage), $(\mathbf{I}+1)^{\text{th}}$ instruction performs step 1
- Non-overlapping use of functional units/storage elements
- Implementation must address the issues due to changes to state by different instructions in the same clock cycle



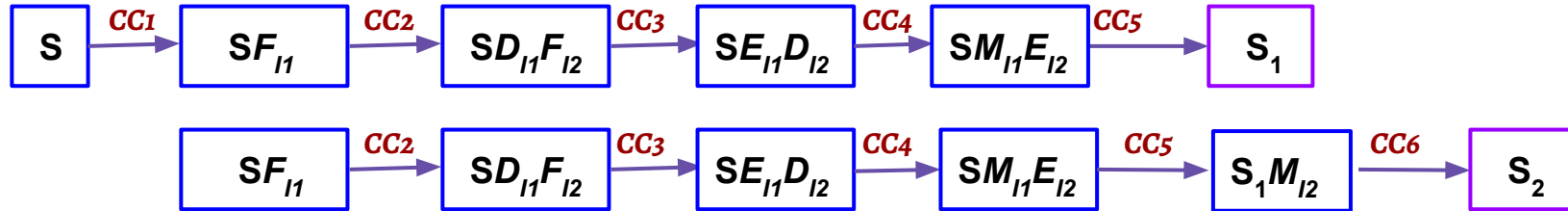
Pipelining overview

- Software visible state transition ($\mathbf{S} \rightarrow \mathbf{Q}$) in $\mathbf{K}$ clock cycles for a given instruction
- While $\mathbf{I}^{\text{th}}$ instruction is in 2nd step (stage), $(\mathbf{I}+\mathbf{1})^{\text{th}}$ instruction performs step 1
- Non-overlapping use of functional units/storage elements
- Implementation must address the issues due to changes to state by different instructions in the same clock cycle



Pipelining overview

- Software visible state transition ($\mathbf{S} \rightarrow \mathbf{Q}$) in $\mathbf{K}$ clock cycles for a given instruction
- While $\mathbf{I}^{\text{th}}$ instruction is in 2nd step (stage), $(\mathbf{I}+1)^{\text{th}}$ instruction performs step 1
- Non-overlapping use of functional units/storage elements
- Implementation must address the issues due to changes to state by different instructions in the same clock cycle



- State overlap: can be useful and create problems at the same time. Example: PC
- Typical implementations copy and carry the required state along the pipeline

Ideal pipeline performance

Operation	Time
Instruction Fetch	200 ps
Register Read/Write	150 ps
ALU Operation	150 ps
Data Access	200 ps

What will be the time taken to execute the workload with a 5-stage pipeline?

- Clock cycle time: Single-cycle = 850 ps, Multi-cycle = 200 ps
- Consider a workload with 10 billion instructions where 40% are R-Type, 10% are Load, 20% are Store and 30% are branch instructions. What will be the execution time in single-cycle and multi-cycle implementations? Single-cycle = 8.5 sec
Multi-cycle = 7.6 sec

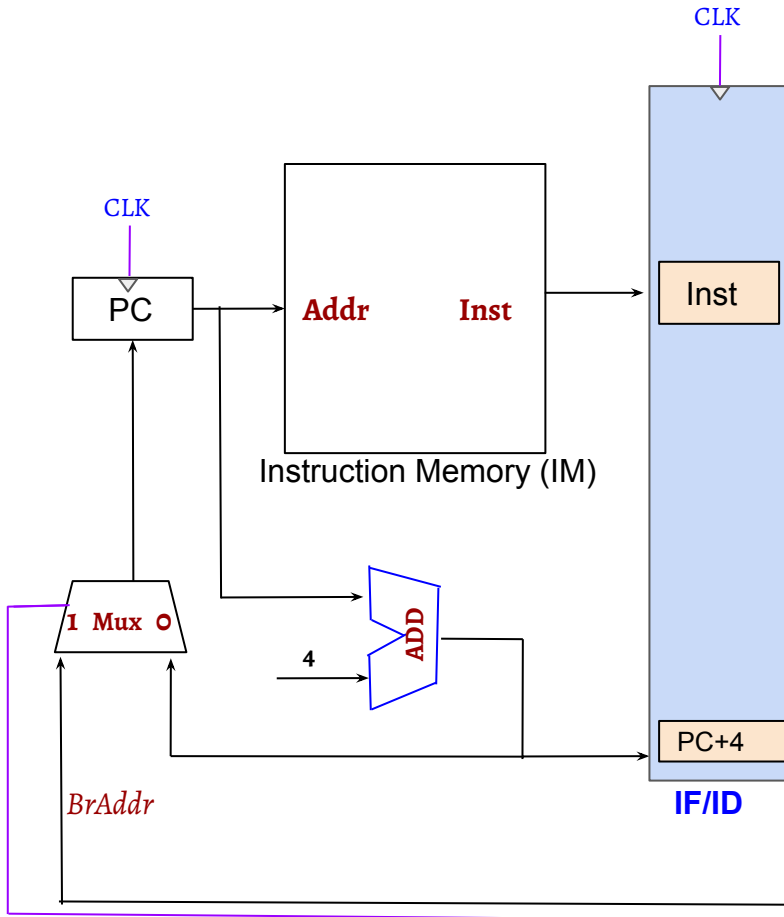
Ideal pipeline performance

Operation	Time
Instruction Fetch	200 ps
Register Read/Write	150 ps
ALU Operation	150 ps
Data Access	200 ps

What will be the time taken to execute the workload with a 5-stage pipeline? 2sec

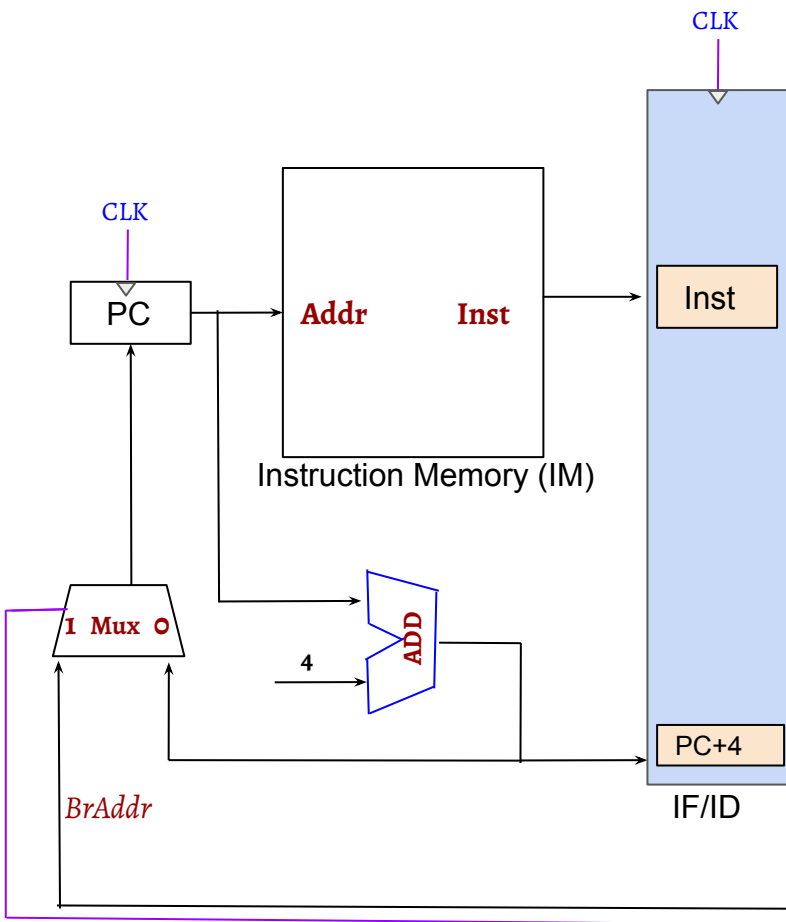
- Clock cycle time: Single-cycle = 850 ps, Multi-cycle = 200 ps
- Consider a workload with 10 billion instructions where 40% are R-Type, 10% are Load, 20% are Store and 30% are branch instructions. What will be the execution time in single-cycle and multi-cycle implementations? Single-cycle = 8.5 sec
Multi-cycle = 7.6 sec

Pipelining: Fetch



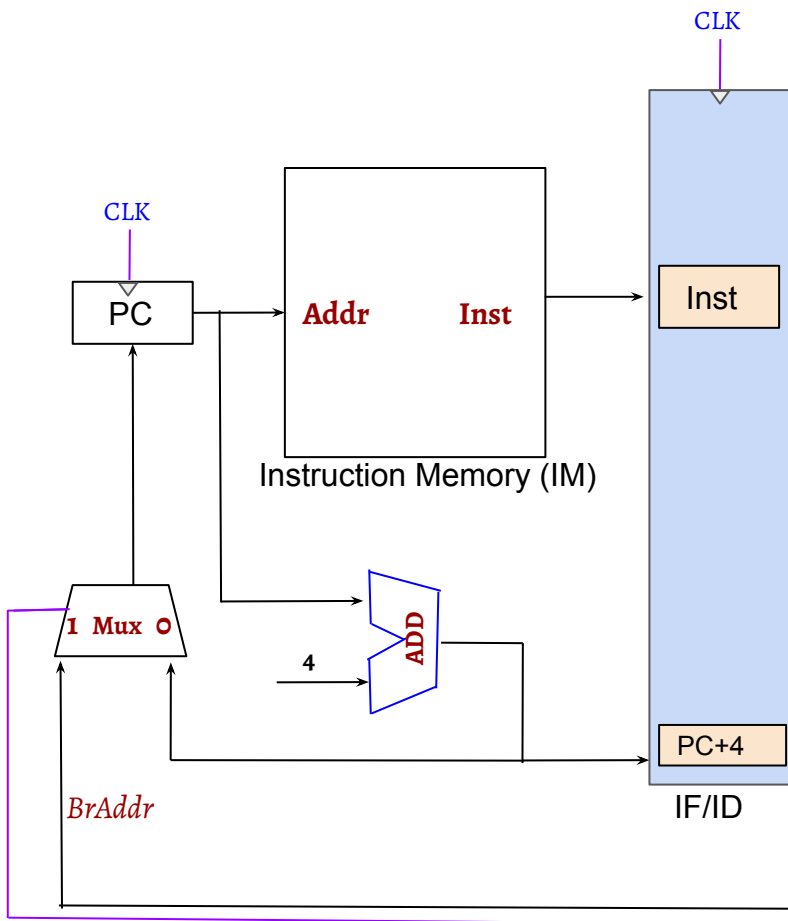
- Fetch is similar to a single cycle operations during the fetch step
- To carry along the required state, new storage elements (pipeline registers) are introduced between stages
- The IF/ID pipeline register is written at the end of fetch cycle
- State elements used: *IM (Read), PC (Read + Write), IF/ID (Write)*

Pipelining: Fetch



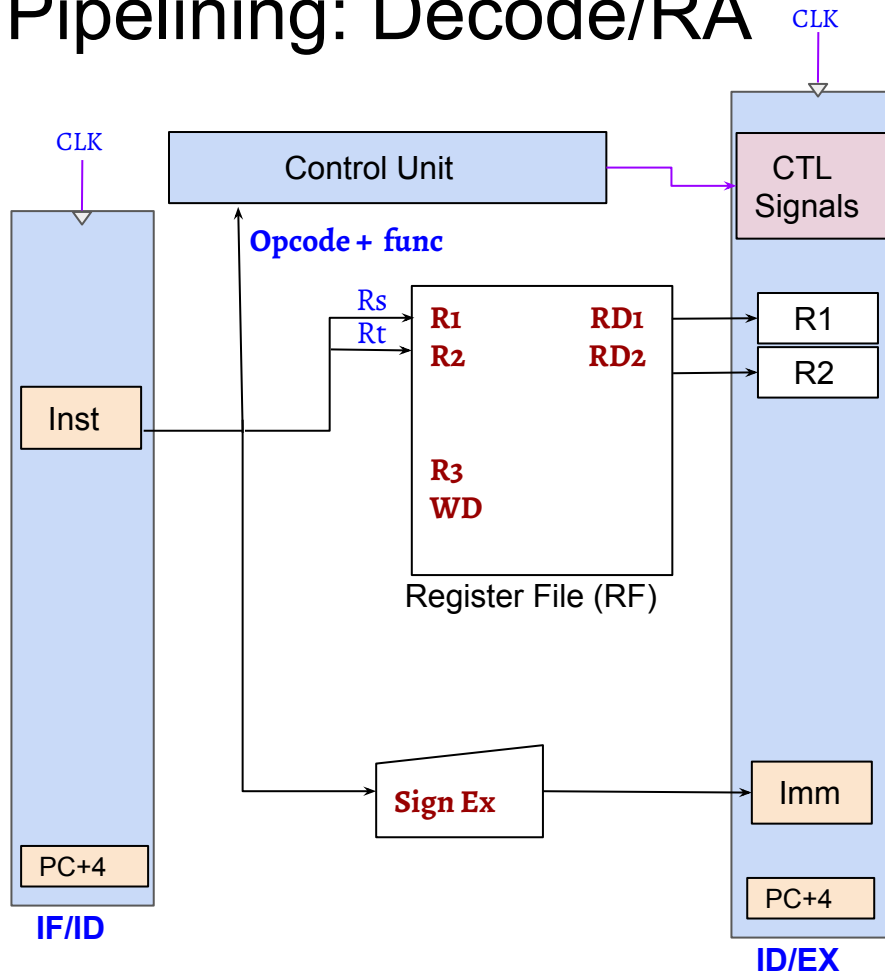
- Why ALU can not be used to increment PC similar to multi-cycle?
- Why IR and PC are required to be carried along?
- How many bits are required for carrying information along?

Pipelining: Fetch



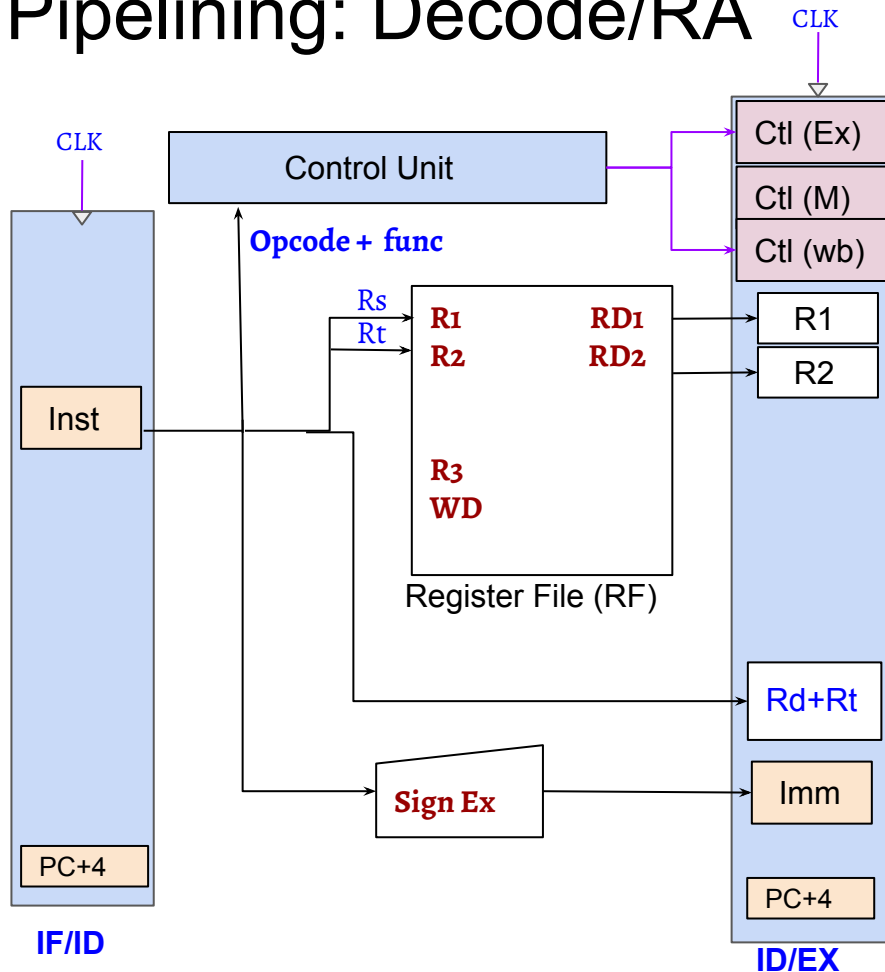
- Why ALU can not be used to increment PC similar to multi-cycle?
 - Can not use ALU because a prev. instruction in Ex/Ad stage may be using
- Why instruction and PC are required to be carried along?
 - Future instructions will modify the PC or the instruction register
- How many pipeline registers (to maintain state) are required?
 - 64 bits assuming PC and Instruction sizes are 32-bits

Pipelining: Decode/RA



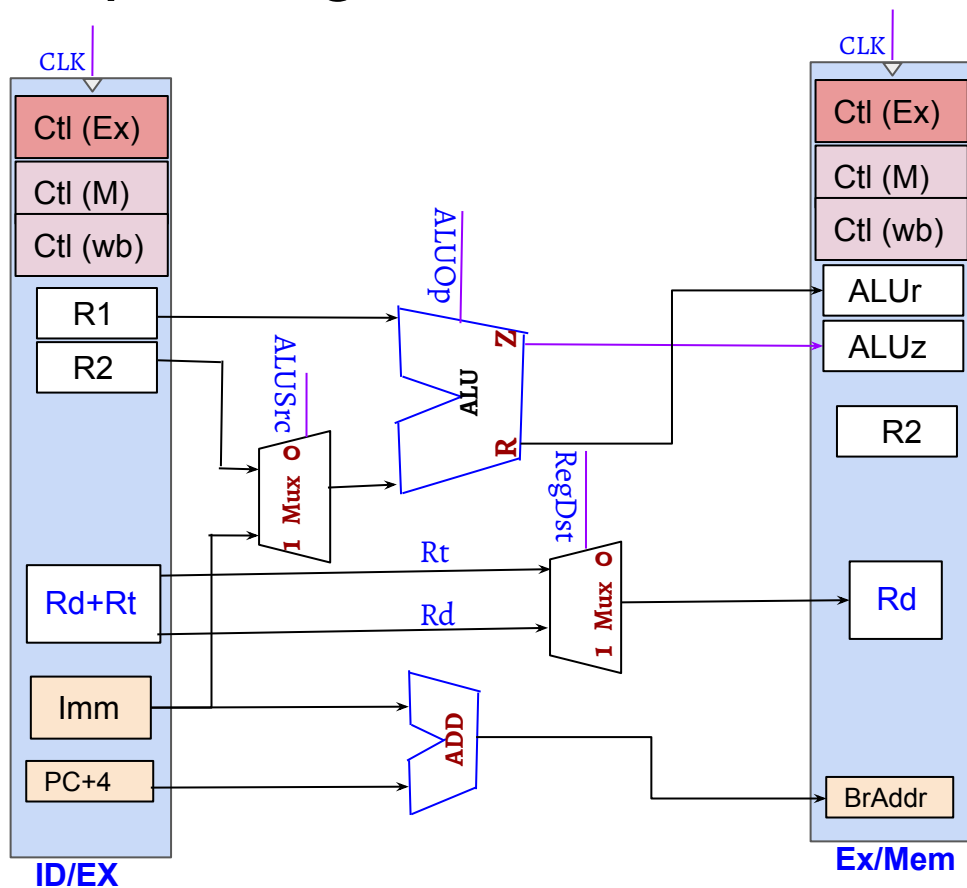
- Control signals are generated based on opcode and function values
- State/FU elements used: *RF (Read), IF/ID (Read), ID/EX (Write)*
- Different signals are required in different stages. How to implement?
- Register and Imm. values are carried along through the pipeline register, why?
- There is a BUG in the implementation!

Pipelining: Decode/RA



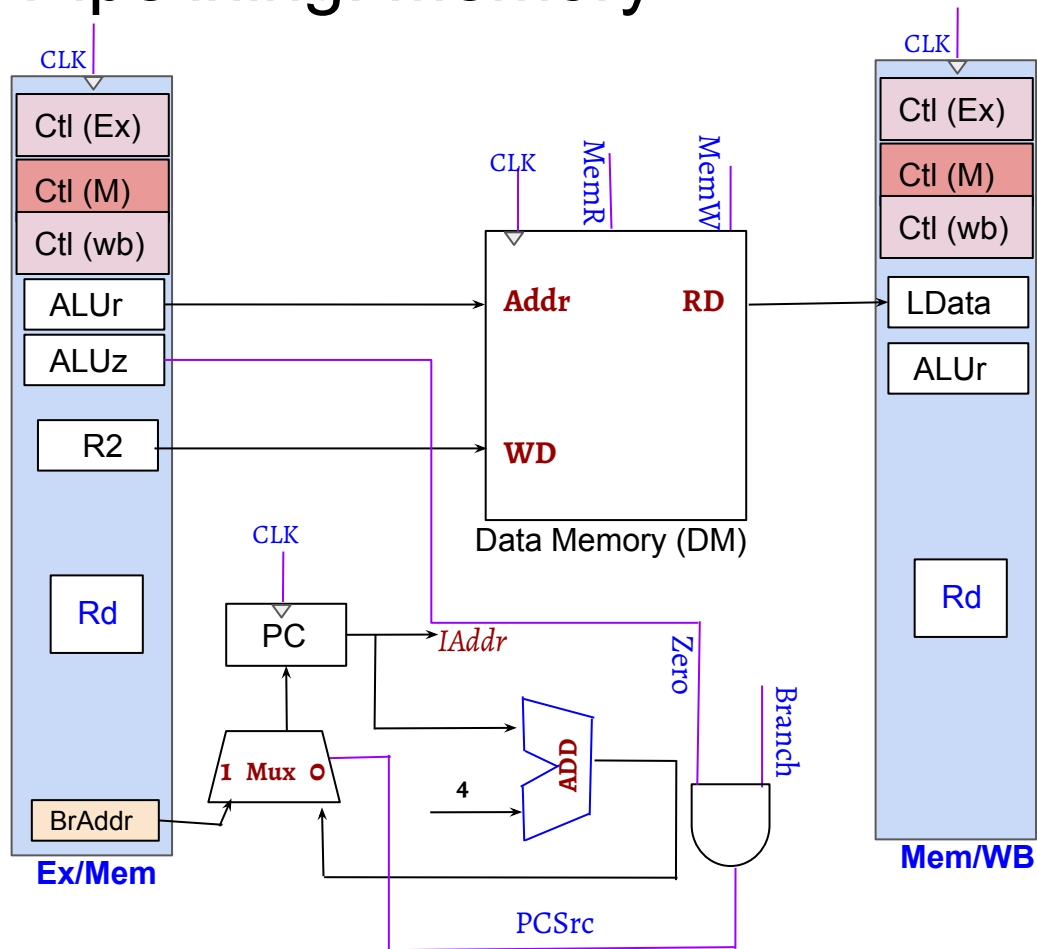
- Control signals are generated based on opcode and function values
- Different signals are required in different stages, How to implement?
 - CU generate per-stage control signals or FSM in the PR to change the value of control signals on each clock
- Register and Imm. values are carried along through the pipeline register, why?
 - Instruction trailing in the pipeline can change the values
- There is a BUG in the implementation!
 - Target and destination register addresses need to be carried along

Pipelining: Execute



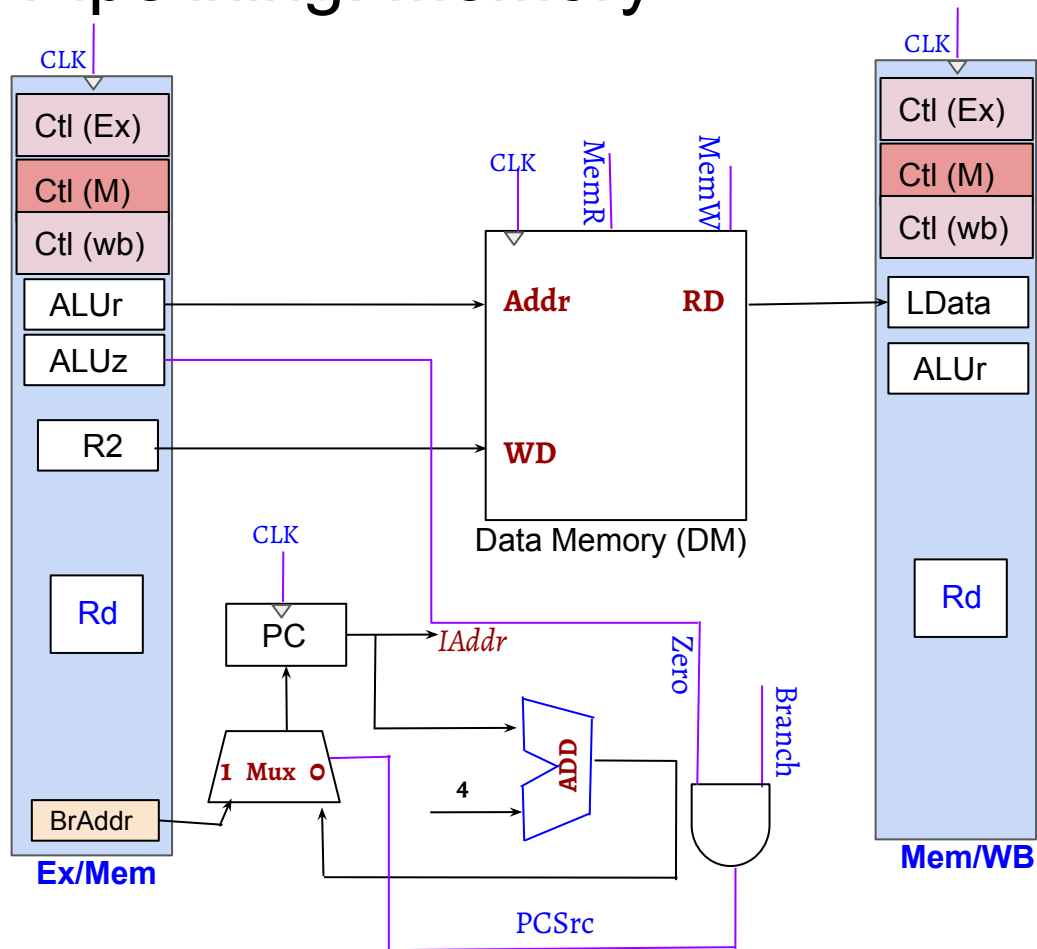
- ALU result (ALUr) is either the address (for LD/ST) or result (for R-Type)
- The destination register address is deduced based on the RegDst control signal value
- $BrAddr = PC + 4 + offset$ where PC is the value when this instruction was fetched
- State/FU elements used: ALU, ID/EX (Read), EX/MA(Write)

Pipelining: Memory



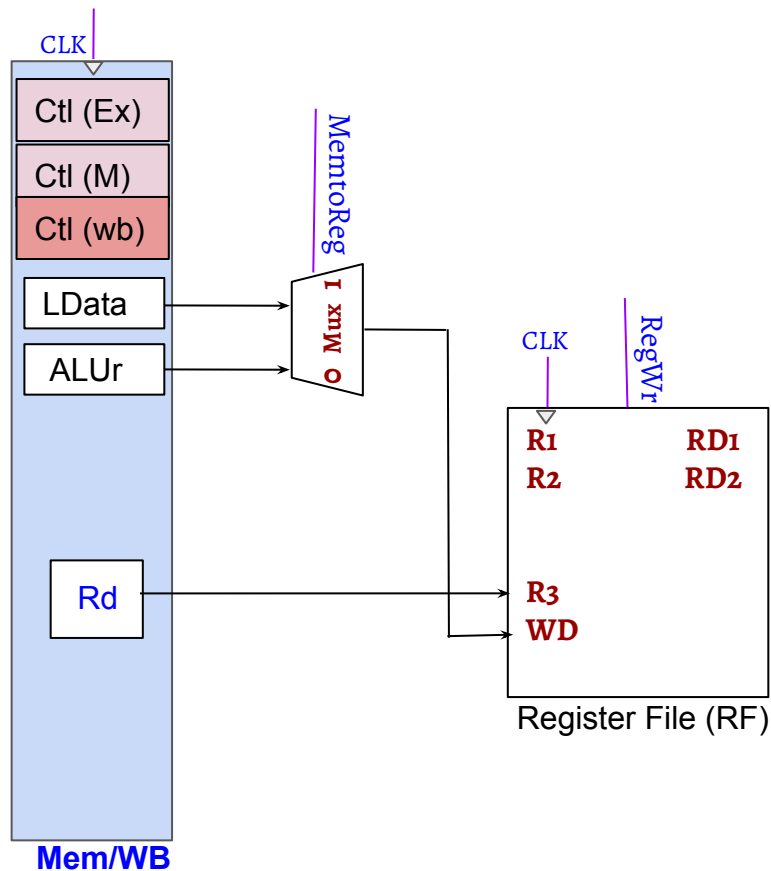
- Result of load instruction is saved in the pipeline register (LData)
- Branch decision taken based on the ALU zero o/p (in PR) and branch control signal
- State/FU elements used: *PC(Write)*, *Memory(Read/Write)*, *EX/MA(Read)*, *Mem/WB(Write)*
- Assume PCSrc = 0, If the value of PC before the fetch of this instruction was **X**, what value will be the written to PC?

Pipelining: Memory



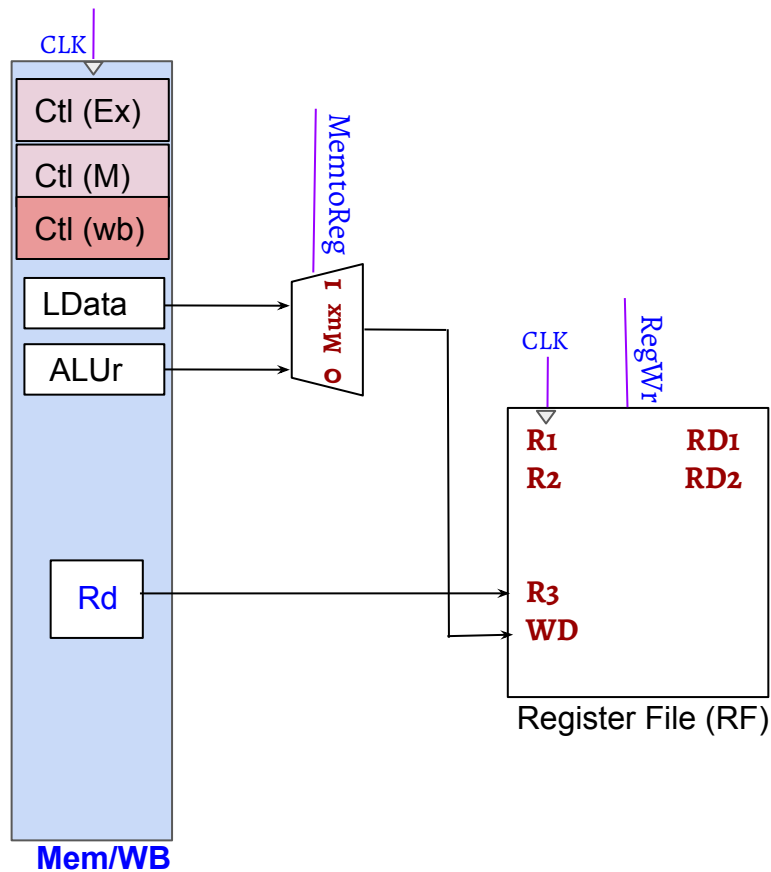
- Result of load instruction is saved in the pipeline register (LData)
- Branch decision taken based on the ALU zero o/p (in PR) and branch control signal
- State/FU elements used: *PC(Write)*, *Memory(Read/Write)*, *EX/MA(Read)*, *Mem/WB(Write)*
- Assume PCSrc = 0, If the value of PC before the fetch of this instruction was **X**, what value will be the written to PC? **X+16**

Pipelining: Write back



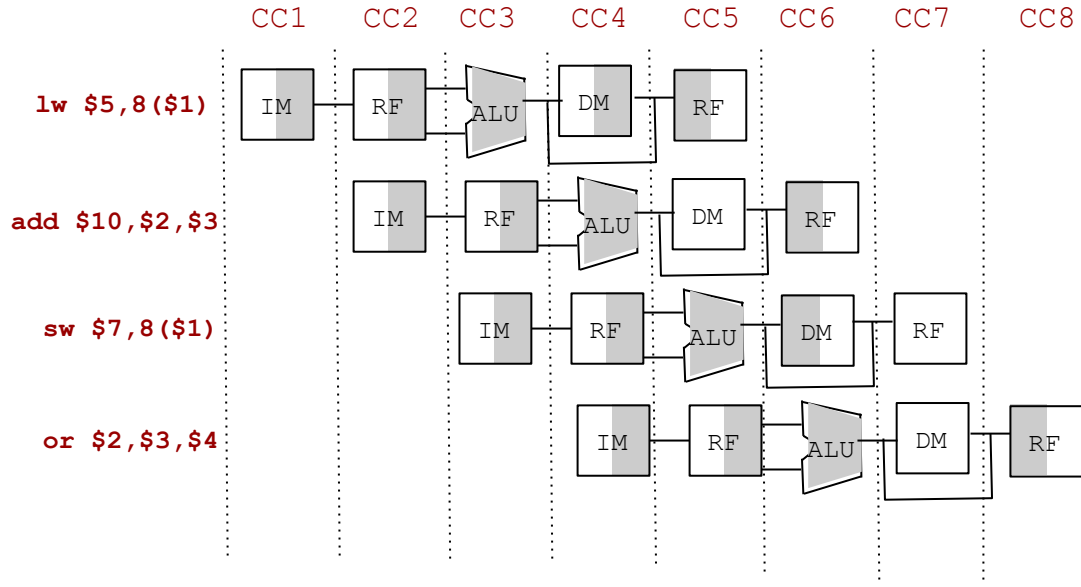
- Depending on the value of MemtoReg, the loaded data (for LD) or ALU result (for R-Type) stored back to the register file
- Pipeline state need not be maintained, the user visible state is updated in this stage
- State/FU elements used: *RF (Write), Mem/WB(Read)*
- If a branch is taken in the previous stage of the pipeline, what would happen to this stage of execution?

Pipelining: Write back



- Depending on the value of MemtoReg, the loaded data (for LD) or ALU result (for R-Type) stored back to the register file
- Pipeline state need not be maintained, the user visible state is updated in this stage
- State/FU elements used: *RF (Write), Mem/WB(Read)*
- If a branch is taken in the previous stage of the pipeline, what would happen to this stage of execution?
 - Complete this stage with additional logic if required (example: implement branch due to exceptions)

Representing the pipeline



- Each stage is represented by the FU/storage element used
- Shaded region shows nature of use—read (right-half), write (left-half)
- For a RF with single read-port, two reads can not happen in the same clock cycle
- For RF, read can happen in the second half of the clock cycle
- Unused FU/storage units are not shaded e.g., DM not used for `or` and `add`

Pipeline is good, but ...

- Implications of pipelining a single cycle implementation into multiple stages
 - Splitting stages for non-overlapped execution, cost vs. performance tradeoff
 - Balancing stages is critical – slowest stage determines clock cycle
 - Additional overhead to maintain pipeline registers

Pipeline is good, but ...

- Implications of pipelining a single cycle implementation into multiple stages
 - Splitting stages for non-overlapped execution, cost vs. performance tradeoff
 - Balancing stages is critical – slowest stage determines clock cycle
 - Additional overhead to maintain pipeline registers
- Even with perfectly balanced pipelines, pipeline can perform poorly because of
 - Data dependence
 - Disruption of sequential execution flow
 - Program structure (if-else, loops, switch-case, procedure call/ret)
 - Exceptions, system calls

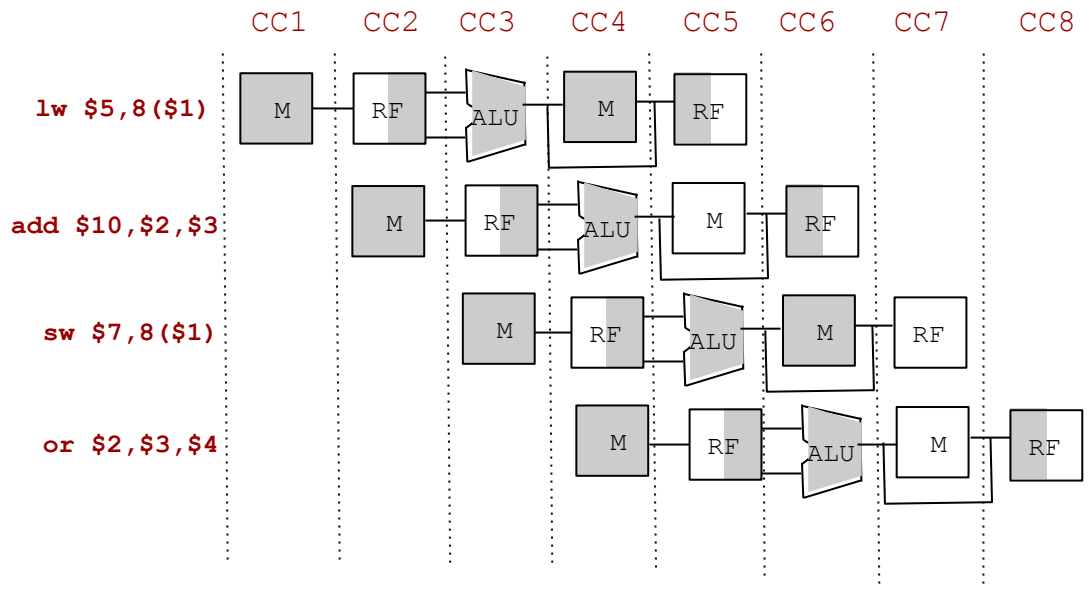
Pipeline is good, but ...

- Implications of pipelining a single cycle implementation into multiple stages
 - **Splitting stages for non-overlapped execution, cost vs. performance tradeoff**
 - Balancing stages is critical – slowest stage determine clock cycle
 - Additional overhead to maintain pipeline registers
- Even with perfectly balanced pipelines, pipeline can perform poorly because of
 - **Data dependence** **Data hazard**
 - **Disruption of sequential execution flow** **Control hazard**
 - Program structure (if-else, loops, switch-case, procedure call/ret)
 - Exceptions, system calls

**Structural
hazard**

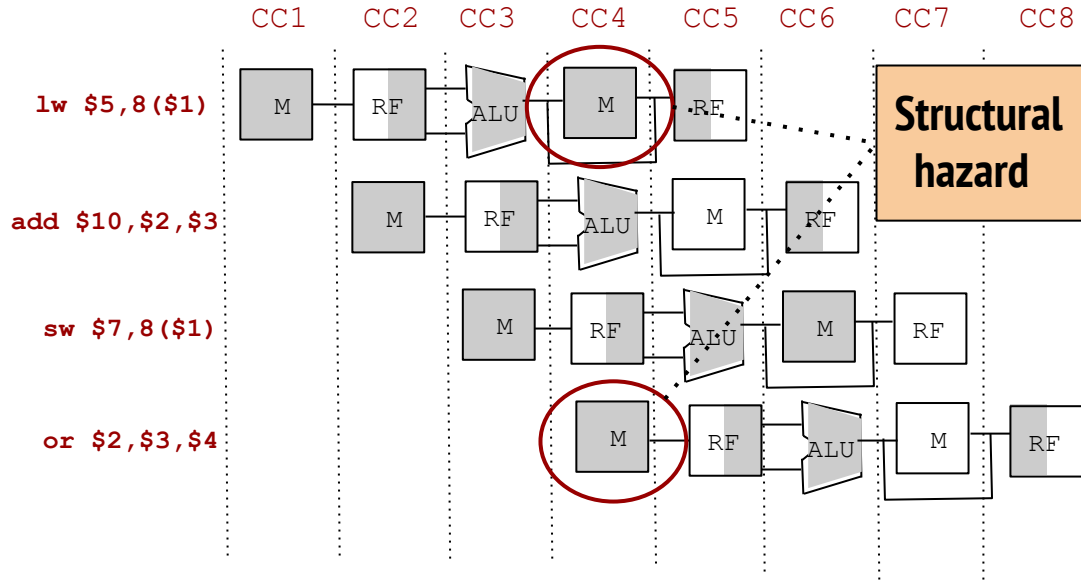
Pipeline Hazard: When an instruction in the *execution stream* can not start execution

Structural hazard



- Assume a unified non-pipelined memory where both data and instructions are accessed
- In CC4, `lw` is in memory stage (reading data from memory), `or` is in IF stage (reading inst. from memory)

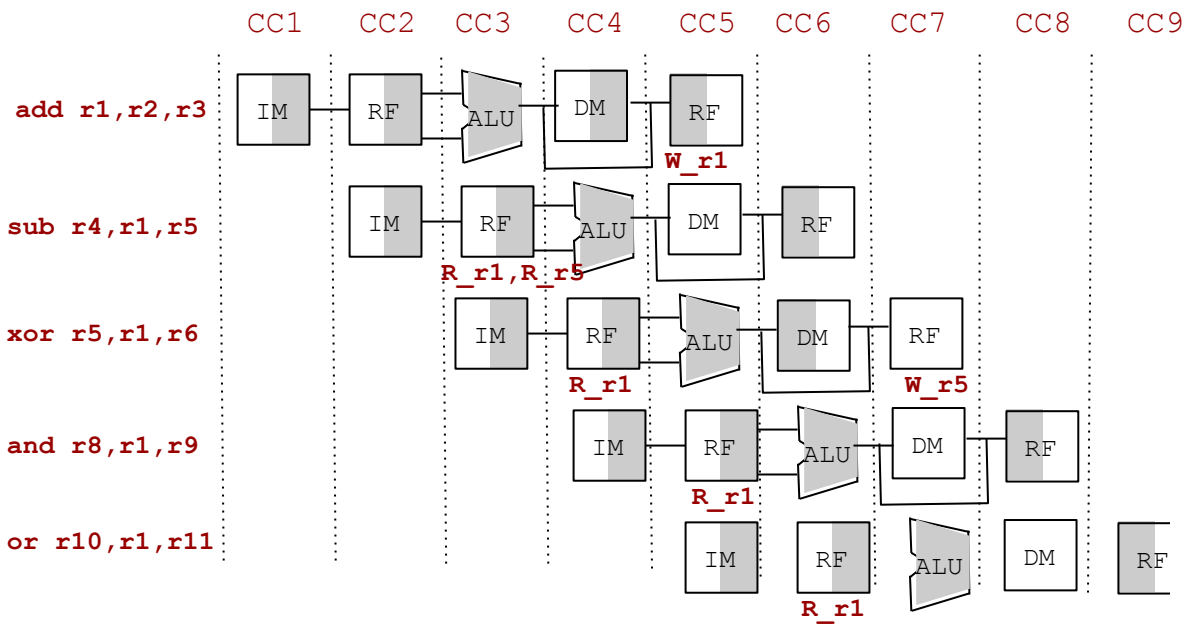
Structural hazard



- Assume a unified non-pipelined memory where both data and instructions are accessed
- In CC4, **lw** is in memory stage (reading data from memory), **or** is in IF stage (reading instruction from memory)

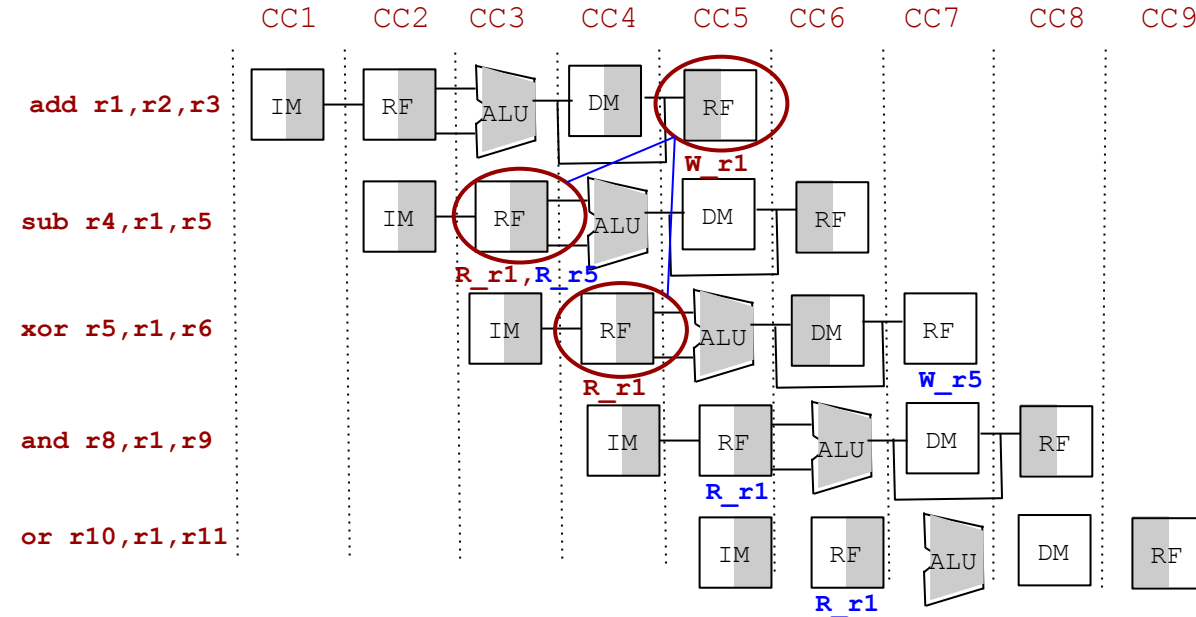
Duplication of resources may help, lead to increased cost and impact resource utilization

Data hazard



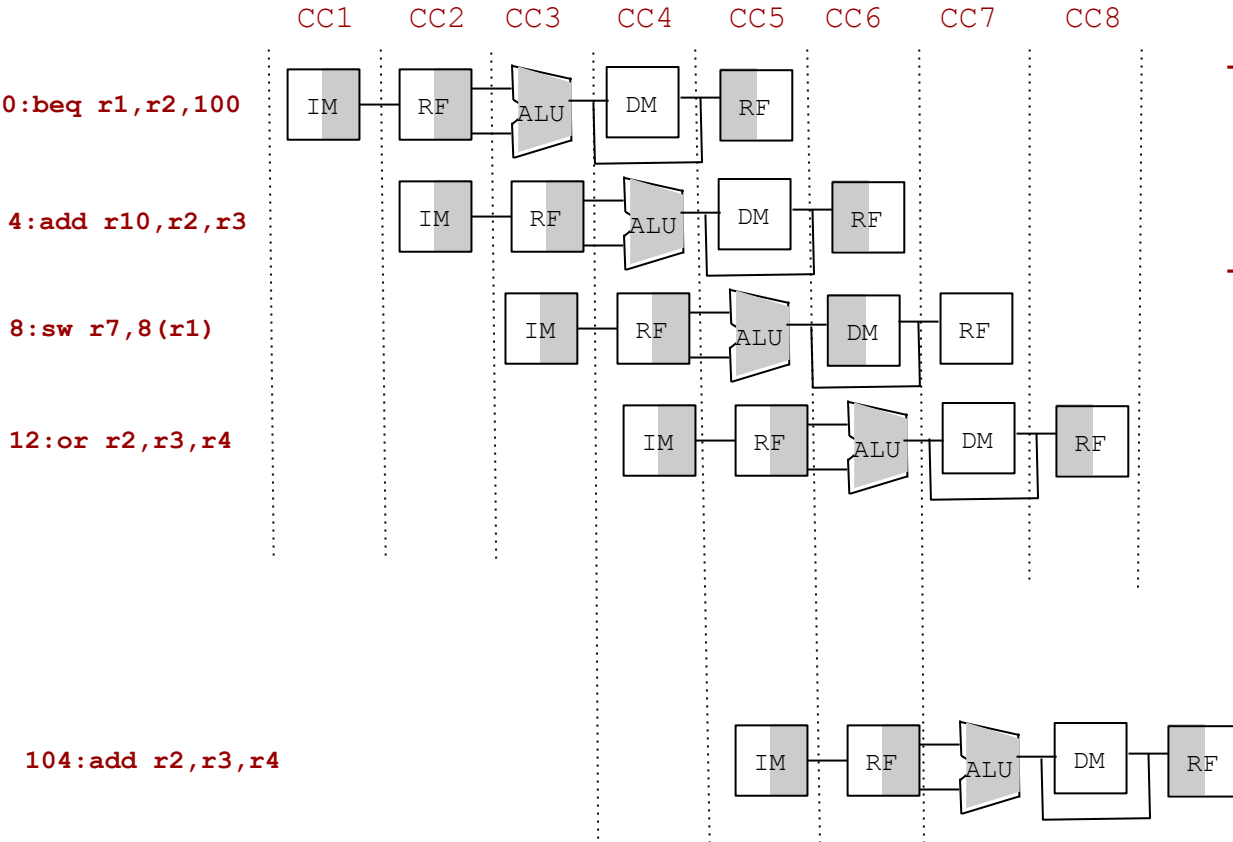
- Register **r1** is written in CC5 (first half of the clock) by **add**
- **r1** is read by **sub**, **xor**, **and**, **or** in the second half of CC3, CC4, CC5 and CC6, respectively
- Which instructions can not proceed?

Data hazard



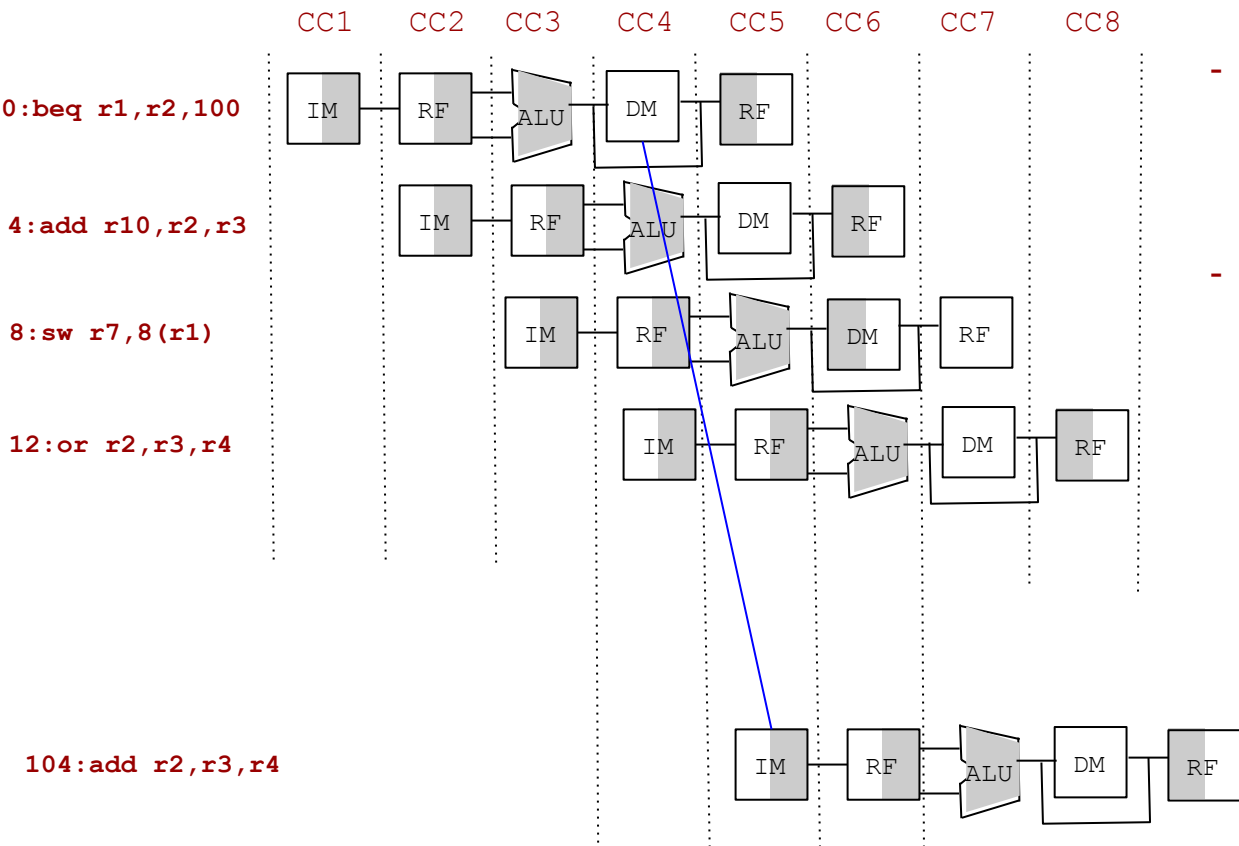
- Register **r1** is written in CC5 (first half of the clock) by **add**
- **r1** is read by **sub**, **xor**, **and**, **or** in the second half of CC3, CC4, CC5 and CC6, respectively
- Which instructions can not proceed? For how long?
- **sub** for 2 cycles and **xor** for 1 cycle

Control hazard (due to branch)



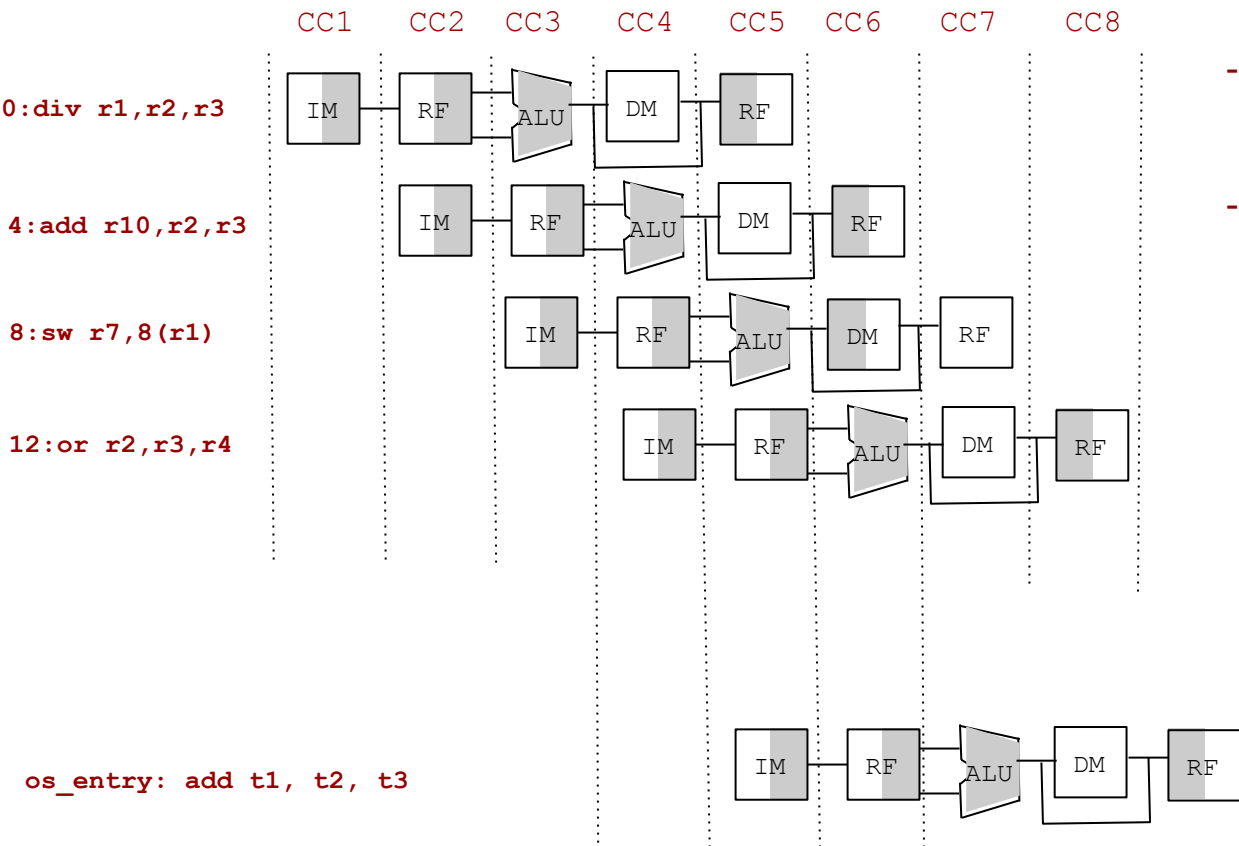
- Assuming $r1=r2$, next instruction will be fetched from memory address 104 in CC5
- What is the impact on pipeline?

Control hazard (due to branch)



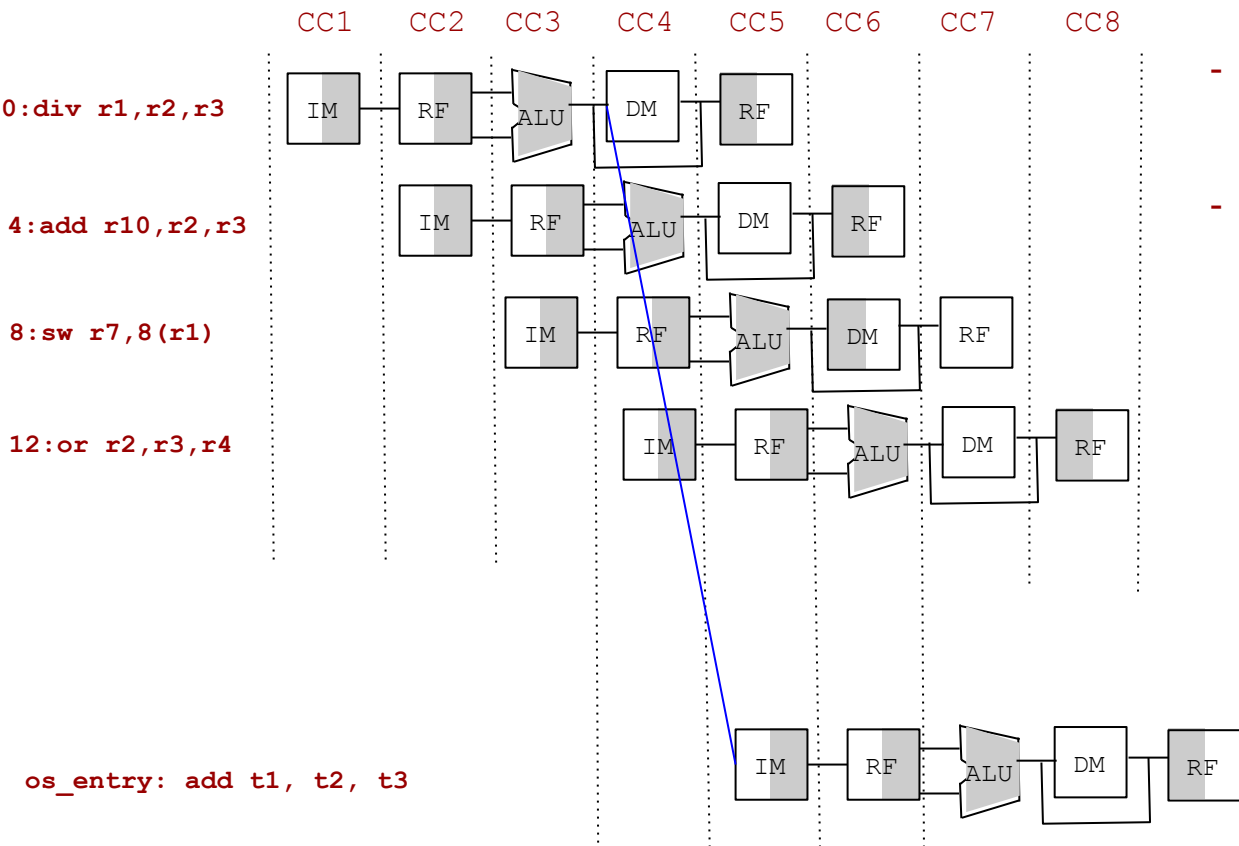
- Assuming $r1=r2$, next instruction will be fetched from memory address 104 in CC5
- What is the impact on pipeline?
 - Three partially executed instructions need to be discarded
 - Frequent reset of the pipeline results in loss of performance

Control hazard (due to exception)



- Assuming $r3=0$, the ALU hardware will raise an exception
- What is the impact on pipeline?

Control hazard (due to exception)



- Assuming $r3=0$, the ALU hardware will raise an exception
- What is the impact on pipeline?
 - Three partially executed instructions need to be *discarded*
 - Additional logic needed to save the state and update the PC with handler address

Pipeline is good, but ...

- Implications of pipelining a single cycle implementation into multiple stages
 - Splitting stages for non-overlapped execution, cost vs. performance tradeoff
 - **Balancing stages is critical – slowest stage determine clock cycle**
 - Additional overhead to maintain pipeline registers
- Even with perfectly balanced pipelines, pipeline can perform poorly because of
 - Data dependence
 - Disruption of sequential execution flow
 - Program structure (`if-else`, `loops`, `switch-case`, `procedure call/return`)
 - Exceptions, system calls

How to balance the pipeline stages?

Is it possible to statically determine the stages?