# ECEC-355: Computer Organization & Architecture

Dr. Anup K. Das

*Electrical and Computer Engineering*

*Drexel University*

## Content

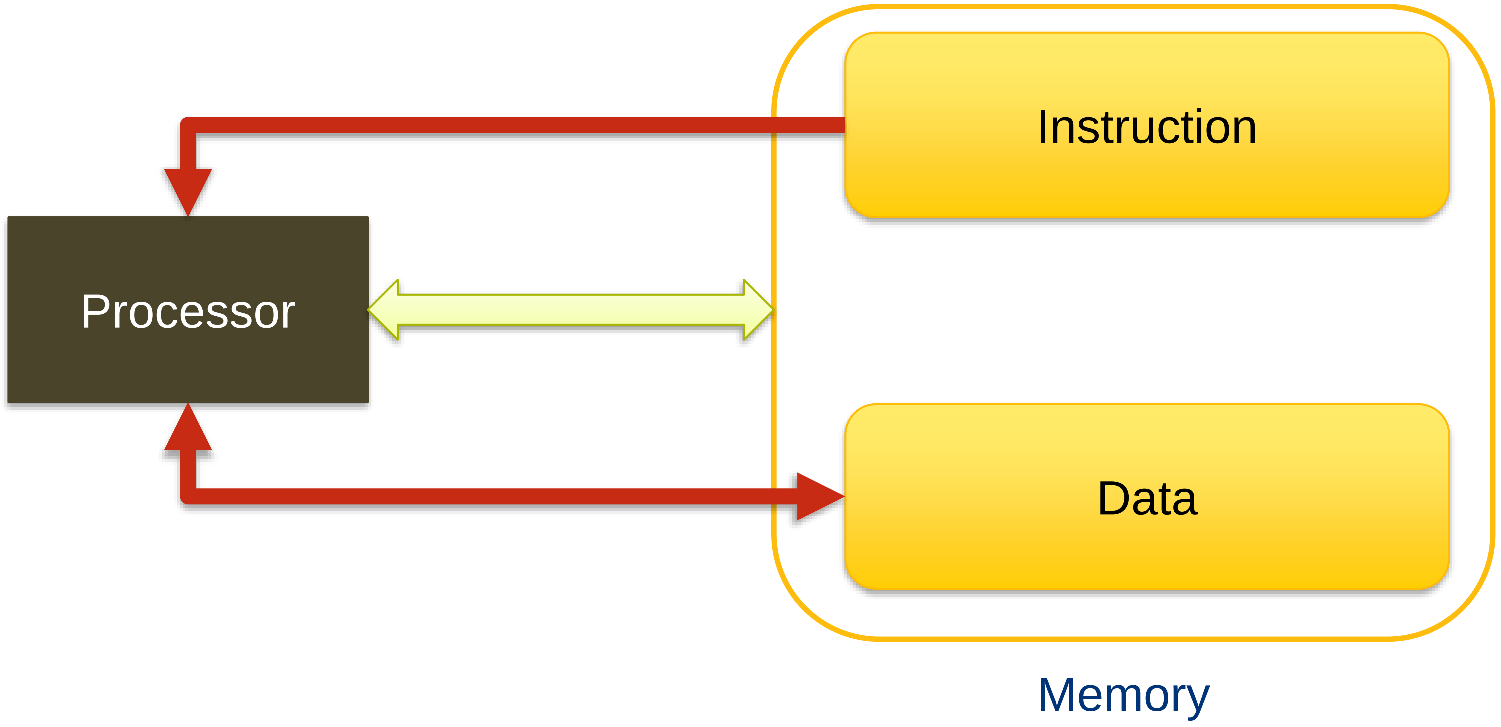
- **Based on Chapter 4 of P&H**

## Designing the microarchitecture

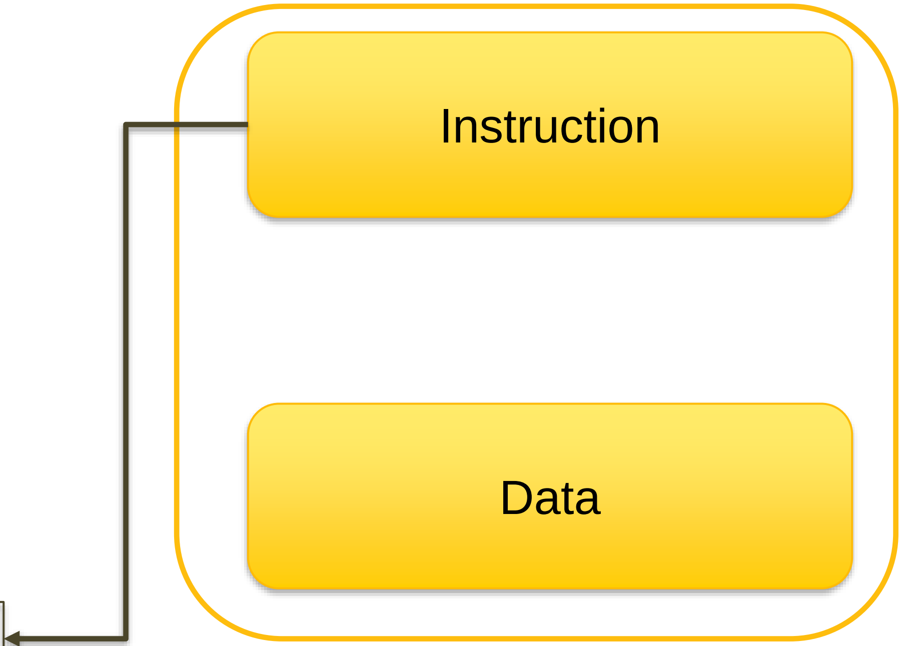
- **Blackbox**
- **Graybox**
- **Whitebox**

# ISA to Microarchitecture (the blackbox)

Observe the direction of the arrows



# Microarchitecture (the blackbox)



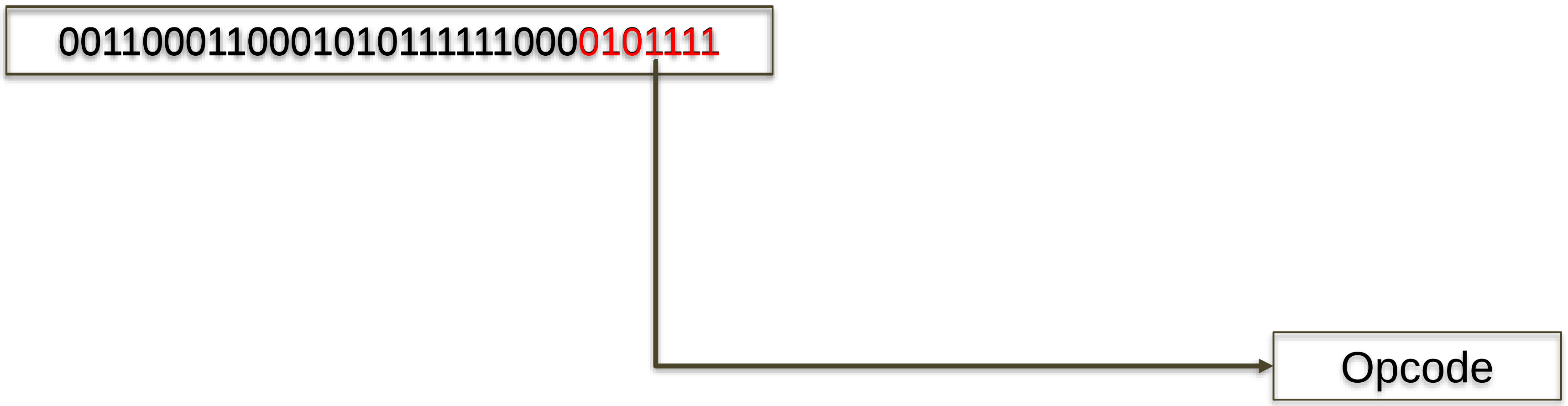
Lets take one such instruction

00110001100010101111110000101111

How many bits?

# Microarchitecture (the blackbox)

| Name (Field Size) | Field 7 bits | 5 bits | 5 bits | 3 bits | 5 bits | 7 bits | Comments |
|---|---|---|---|---|---|---|---|
| R-type | funct7 | rs2 | rs1 | funct3 | rd | opcode | Arithmetic instruction format |
| I-type | immediate[11:0] | | rs1 | funct3 | rd | opcode | Loads & immediate arithmetic |
| S-type | immed[11:5] | rs2 | rs1 | funct3 | immed[4:0] | opcode | Stores |
| SB-type | immed[12,10:5] | rs2 | rs1 | funct3 | immed[4:1,11] | opcode | Conditional branch format |
| UJ-type | immediate[20,10:1,11,19:12] | | | | rd | opcode | Unconditional jump format |
| U-type | immediate[31:12] | | | | rd | opcode | Upper immediate format |



## Microarchitecture (the blackbox)

- **What is opcode?**
  - It tells you what is going to be the operation

## RISC-V-Opcodes

| Operations | Opcode |
|---|---|
| add | 0110011 |
| addi | 0010011 |
| .. | |
| ... | |
| … | |
| … | |
| … | |

What do the processor do?

Performs some operation on data

How to know which operation to perform?

From the opcode

## Microarchitecture (the graybox)

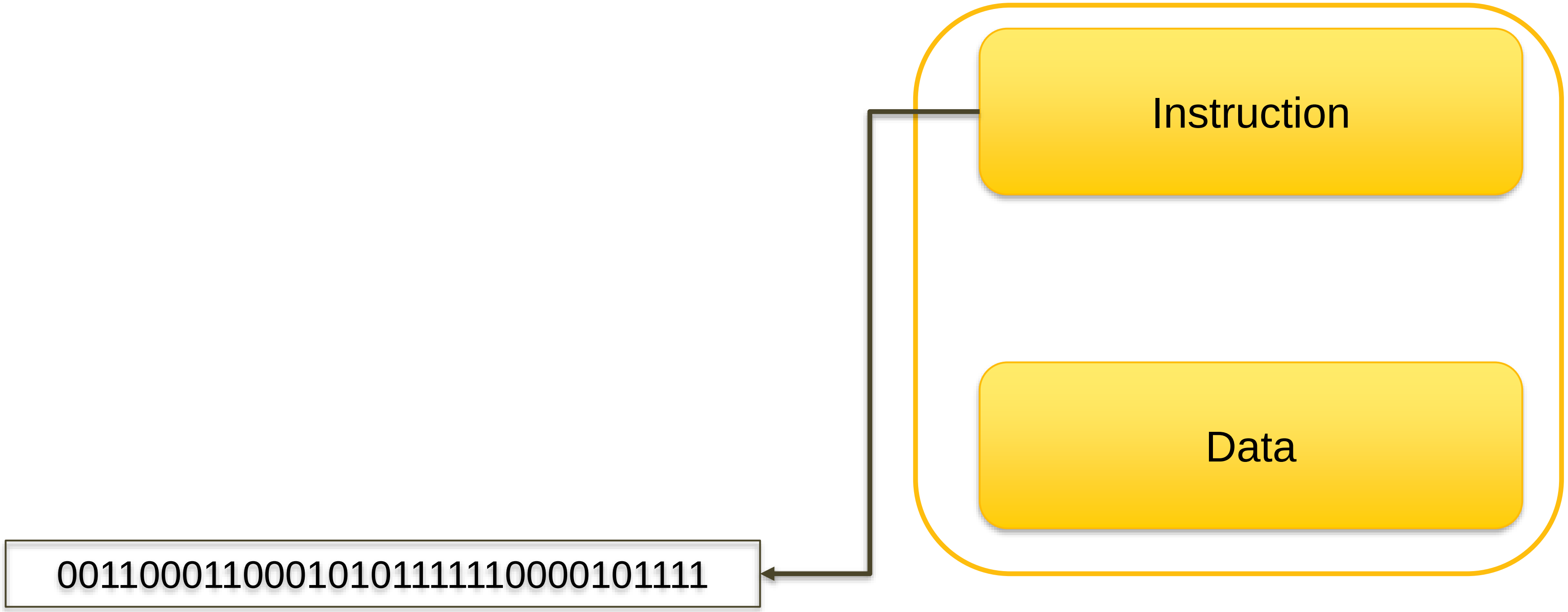
- **Once we have retrieved an instruction, we need to decode the opcode**

- **Steps so far**
  - Fetch the instruction from instruction memory
  - Decode the opcode of the instruction
  - What is the next step?
    - Perform the desired operation

- **Microarchitectural steps for instruction processing**
  - Fetch → Decode → Execute → …

# Microarchitecture (the graybox)

- **Before we move on**
  - Few quick notes on the instruction fetch



## Microarchitecture (the graybox)

- **Recollect: instructions need to be fetched sequentially (one after another)**
  - Control flow / Von-Neuman model
  - Unless you encounter conditional instructions
  - After executing the conditional instruction, you need to go back to the point where you were before to continue executing the next instruction

- **You need a register to keep track of the instruction you are executing**
  - Program counter register
  - This is the first component of your microarchitecture

## Microarchitecture (the graybox)

- **Recap of memory organization**

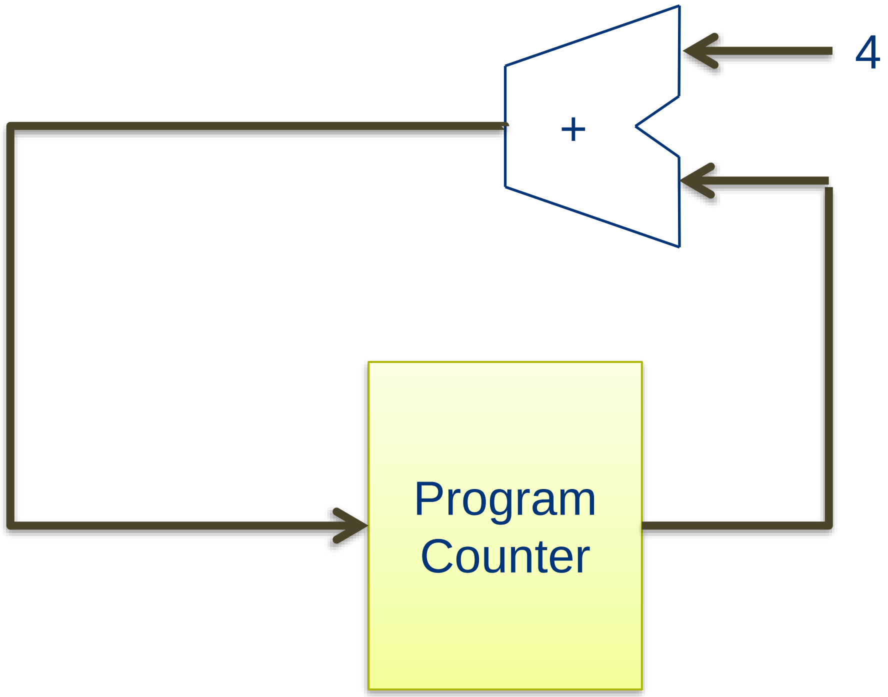
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | a | a+1 | a+2 | a+3 | a+4 | a+5 | a+6 | a+7 |
| a+8 | a+8 | a+9 | a+10 | a+11 | **a+12** | **a+13** | **a+14** | **a+15** |
| a+16 | **a+16** | **a+17** | **a+18** | **a+19** | **a+20** | **a+21** | **a+22** | **a+23** |
| a+24 | **a+24** | **a+25** | **a+26** | **a+27** | **a+28** | | | |
| ⋮ | | | | | | | | |
| | | | | | | | | |

- **Remember each row is a double word (64 bits)**
  - Each instruction is 32 bits (occupies 4 blocks in a double word)
  - Each new instruction is offset by 4 (a, a+4, a+8, …)

## Microarchitecture (the graybox)

- **Instructions**
  - Sequential
  - Control
- **Lets first focus on sequential instructions**
  - Instructions that are fetched and executed one after the other
  - Control instructions jumps to an specified location in the code
    - if-else, while, for, etc.
    - Conditional/unconditional jumps (jal,jar,beq,etc.)
- **For sequential instructions**
  - Next instruction address = current instruction address + 4
  - We have seen this in the last slide!

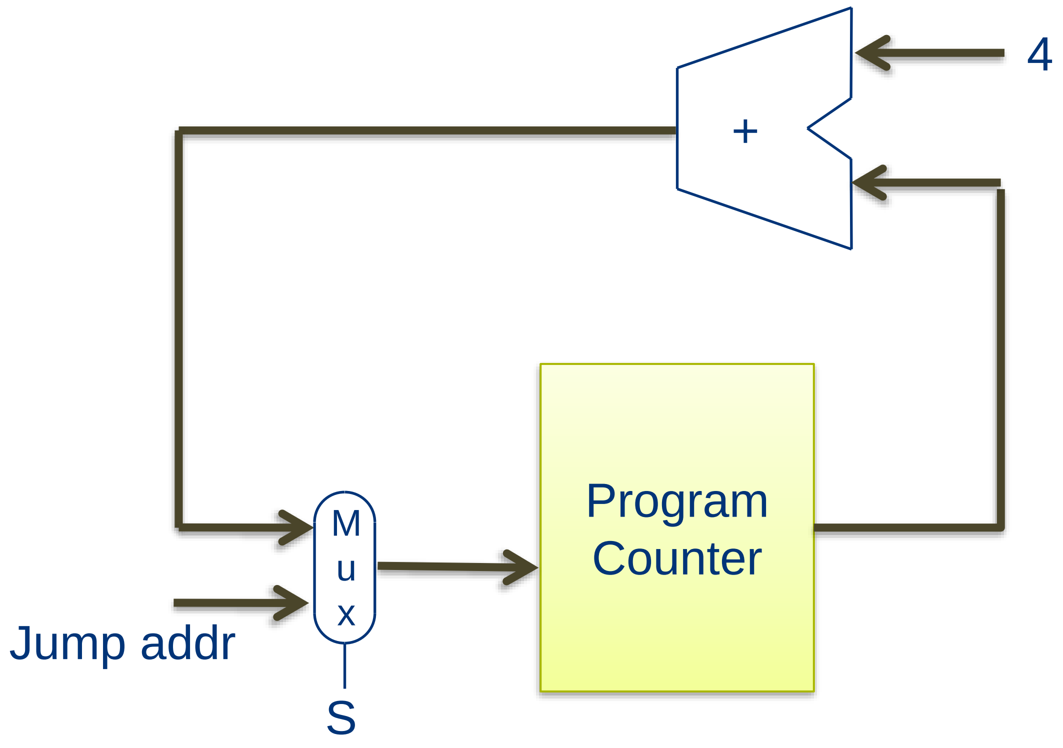
# Microarchitecture (the graybox)



This is for the sequential part

We still need to have some logic for the conditional instructions

# Microarchitecture (the graybox)

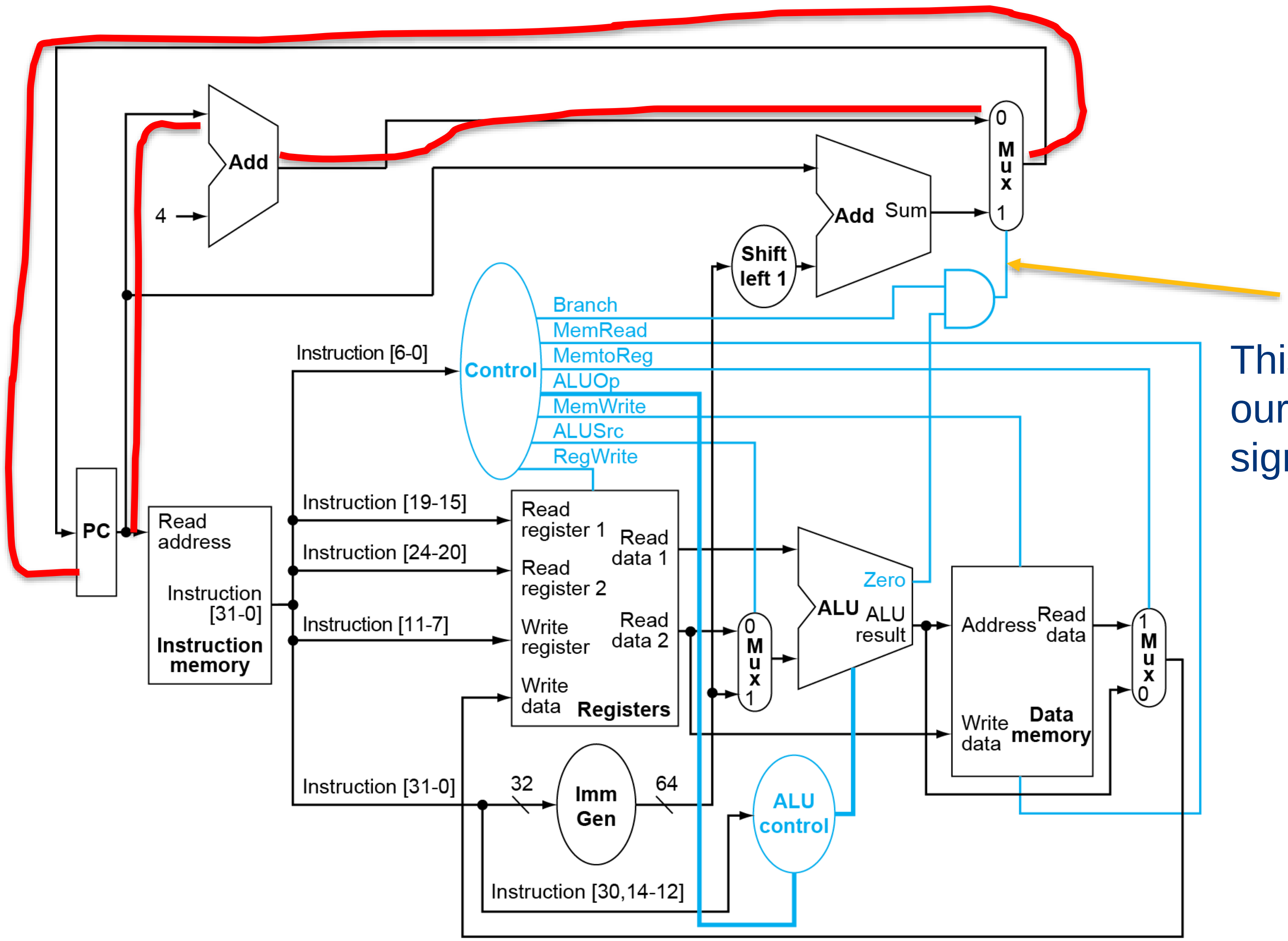


This is for the sequential part

We still need to have some logic for the conditional instructions

If S = 0:  pc = pc + 4
If S = 1:  pc = jump addr

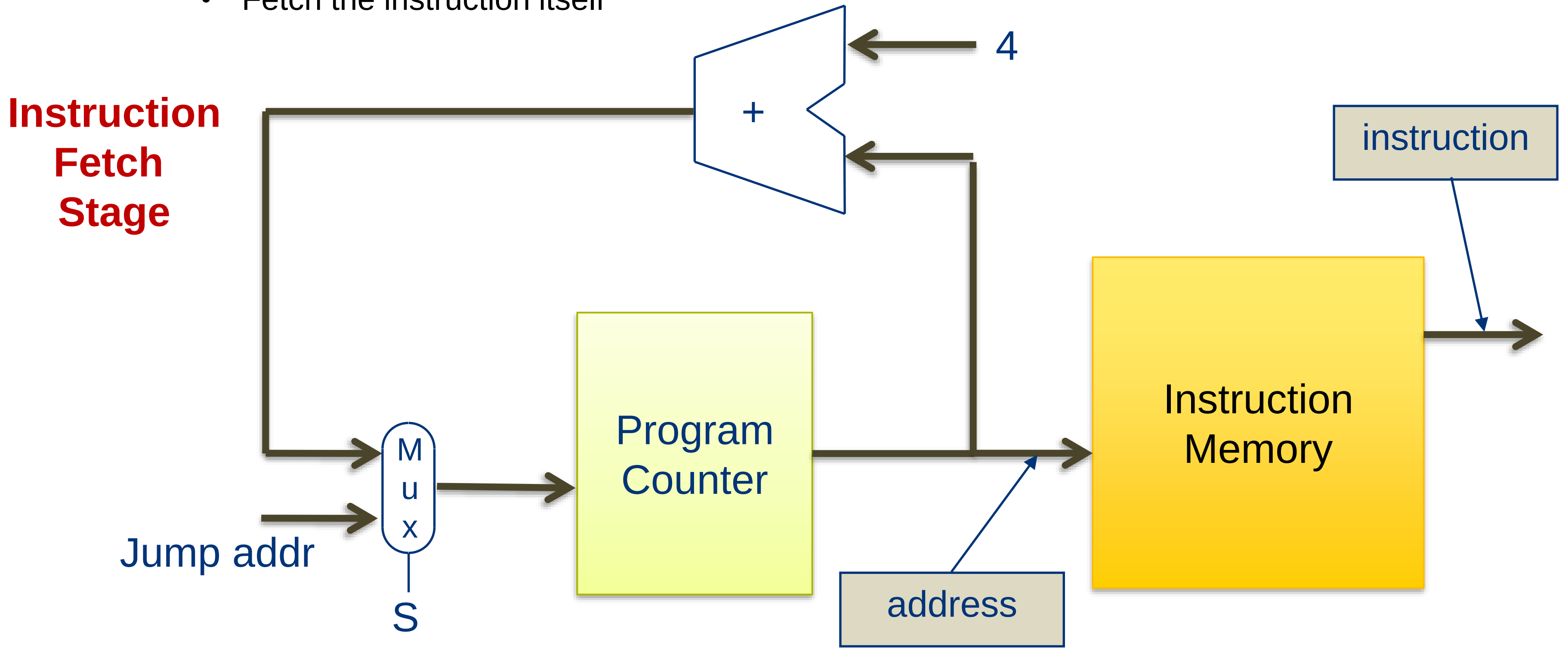
# A quick look at the Microarchitecture



This is our select signal S

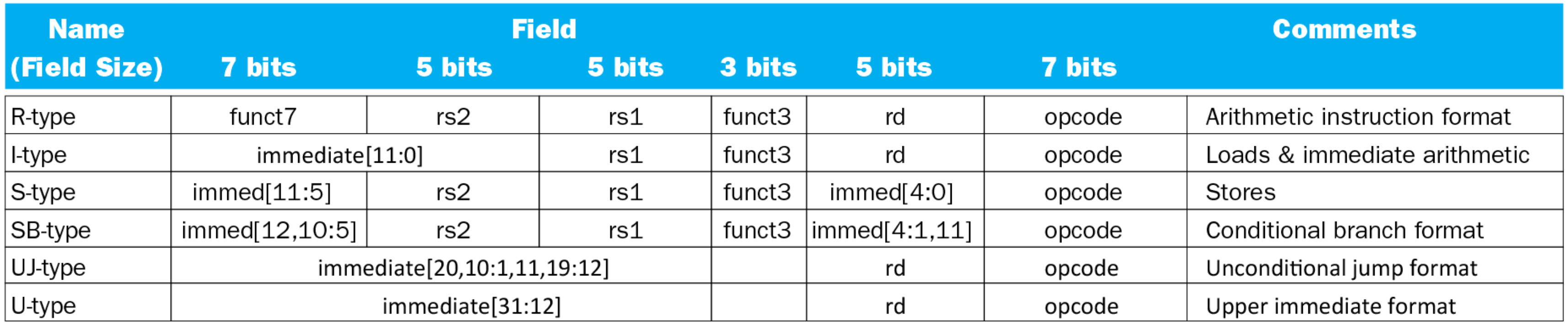
# Microarchitecture (Gray to White)

- **So we have the logic for fetching the address of the next instruction**
  - What to do next?
    - Fetch the instruction itself



# Microarchitecture (Gray to White)

- **You have got the instruction, what to do next?**
  - Decode
  - Extract certain fields from the instruction to
    - Find out what to do, i.e., operations
    - Where to do it to, i.e., operands

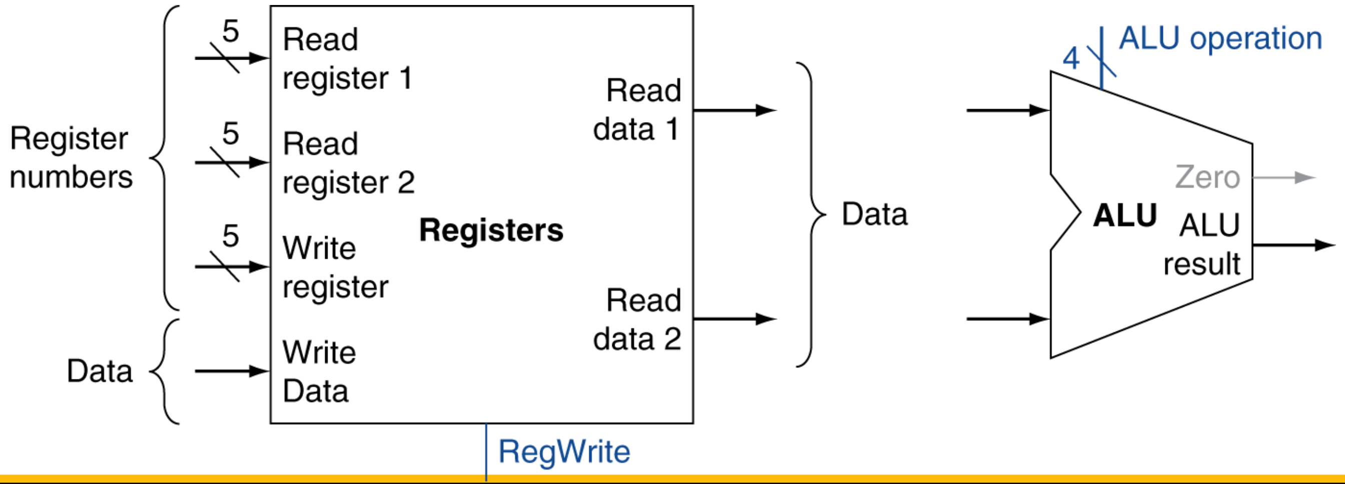


| Name | Field | | | | | | Comments |
|---|---|---|---|---|---|---|---|
| (Field Size) | 7 bits | 5 bits | 5 bits | 3 bits | 5 bits | 7 bits | |
| R-type | funct7 | rs2 | rs1 | funct3 | rd | opcode | Arithmetic instruction format |
| I-type | immediate[11:0] | | rs1 | funct3 | rd | opcode | Loads & immediate arithmetic |
| S-type | immed[11:5] | rs2 | rs1 | funct3 | immed[4:0] | opcode | Stores |
| SB-type | immed[12,10:5] | rs2 | rs1 | funct3 | immed[4:1,11] | opcode | Conditional branch format |
| UJ-type | immediate[20,10:1,11,19:12] | | | | rd | opcode | Unconditional jump format |
| U-type | immediate[31:12] | | | | rd | opcode | Upper immediate format |

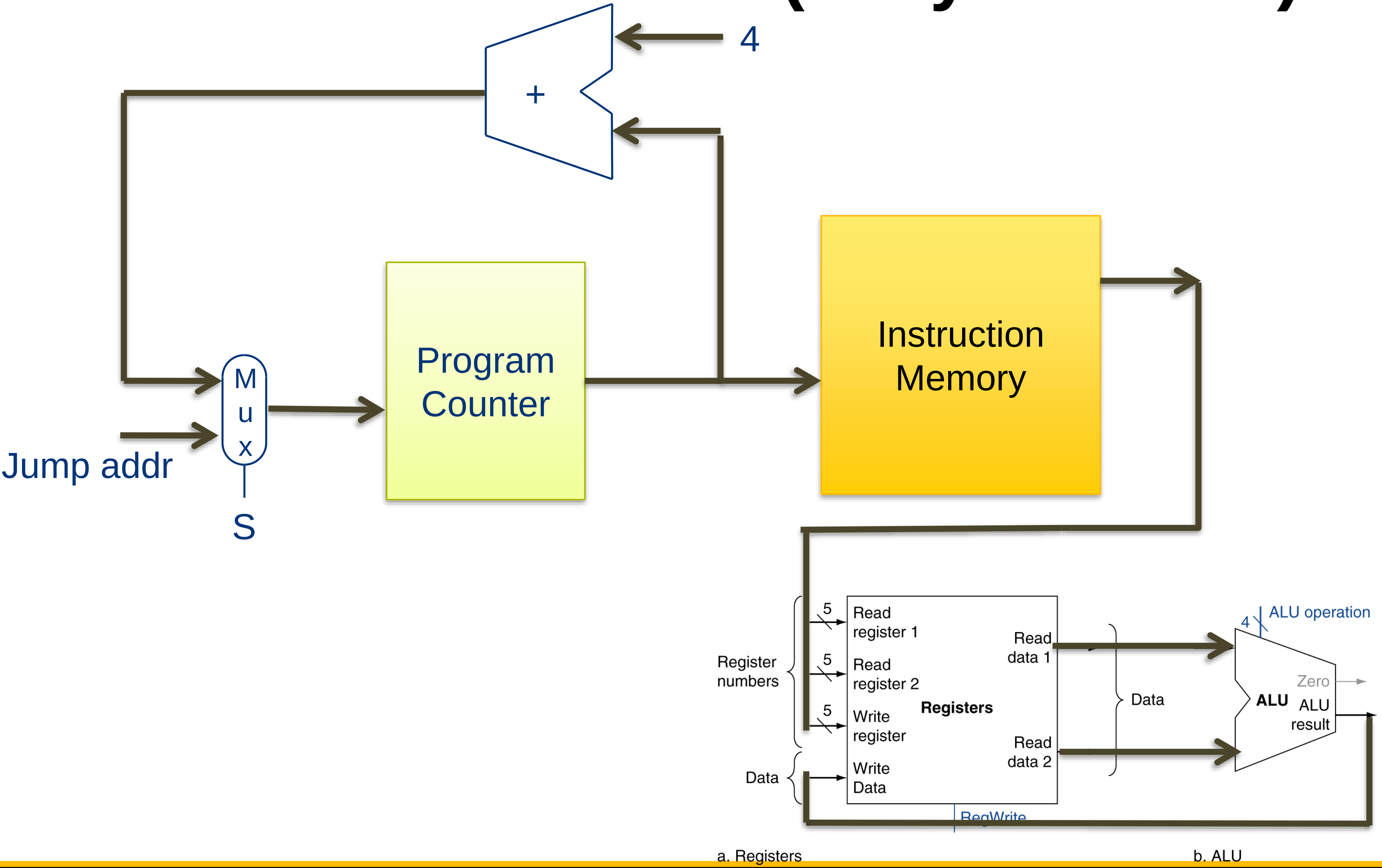
# Microarchitecture (Gray to White)

| Name | Field | | | | | | Comments |
|---|---|---|---|---|---|---|---|
| (Field Size) | 7 bits | 5 bits | 5 bits | 3 bits | 5 bits | 7 bits | |
| R-type | funct7 | rs2 | rs1 | funct3 | rd | opcode | Arithmetic instruction format |
| I-type | immediate[11:0] | | rs1 | funct3 | rd | opcode | Loads & immediate arithmetic |
| S-type | immed[11:5] | rs2 | rs1 | funct3 | immed[4:0] | opcode | Stores |
| SB-type | immed[12,10:5] | rs2 | rs1 | funct3 | immed[4:1,11] | opcode | Conditional branch format |
| UJ-type | immediate[20,10:1,11,19:12] | | | | rd | opcode | Unconditional jump format |
| U-type | immediate[31:12] | | | | rd | opcode | Upper immediate format |

- **R-type Instructions**
  - Read two register operands
  - Perform arithmetic/logical operation
  - Write register result

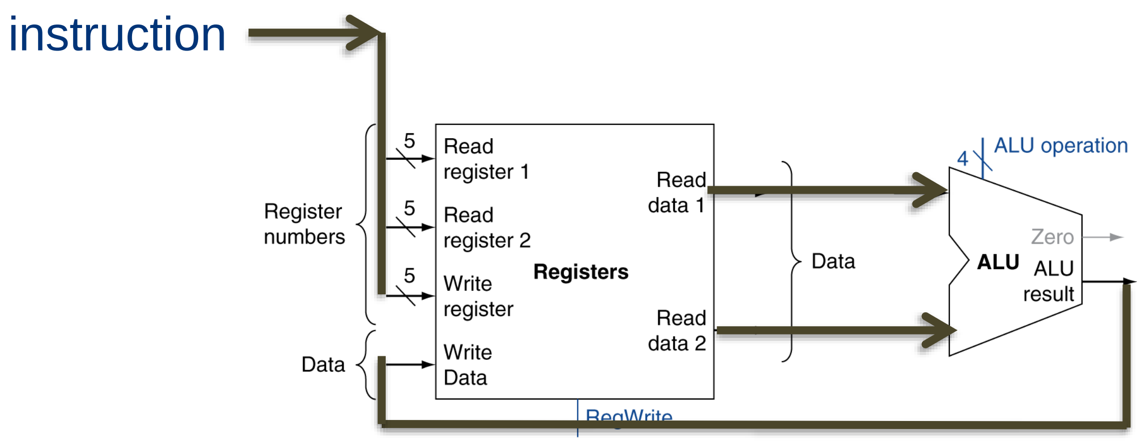


# Microarchitecture (Gray to White)



# Microarchitecture (Gray to White)

- **Every instruction must have fetch unit**
  - What differs is how the instruction is decoded
  - For simplicity we will skip the drawing the fetch unit

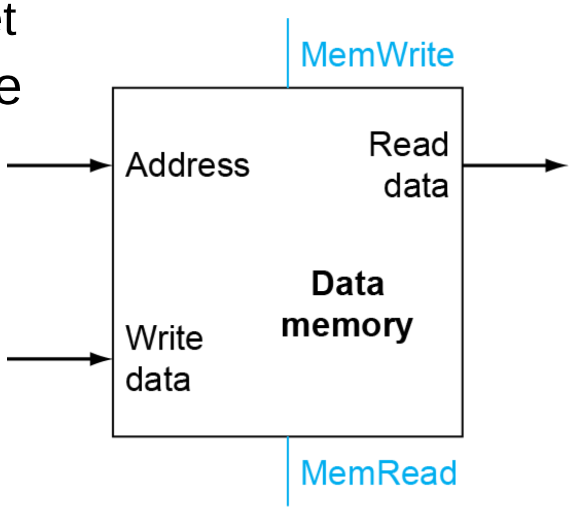


a. Registers b. ALU

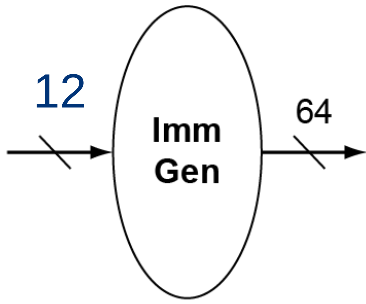
# Microarchitecture (Gray to White)

| Name (Field Size) | Field 7 bits | 5 bits | 5 bits | 3 bits | 5 bits | 7 bits | Comments |
|---|---|---|---|---|---|---|---|
| R-type | funct7 | rs2 | rs1 | funct3 | rd | opcode | Arithmetic instruction format |
| I-type | immediate[11:0] | | rs1 | funct3 | rd | opcode | Loads & immediate arithmetic |
| S-type | immed[11:5] | rs2 | rs1 | funct3 | immed[4:0] | opcode | Stores |
| SB-type | immed[12,10:5] | rs2 | rs1 | funct3 | immed[4:1,11] | opcode | Conditional branch format |
| UJ-type | immediate[20,10:1,11,19:12] | | | | rd | opcode | Unconditional jump format |
| U-type | immediate[31:12] | | | | rd | opcode | Upper immediate format |

- **Load/Store Instructions**
  - Read register operands
  - Calculate address using 12-bit offset
    - Use ALU, but sign-extend offset
  - Load: Read memory and update register
  - Store: Write register value to memory

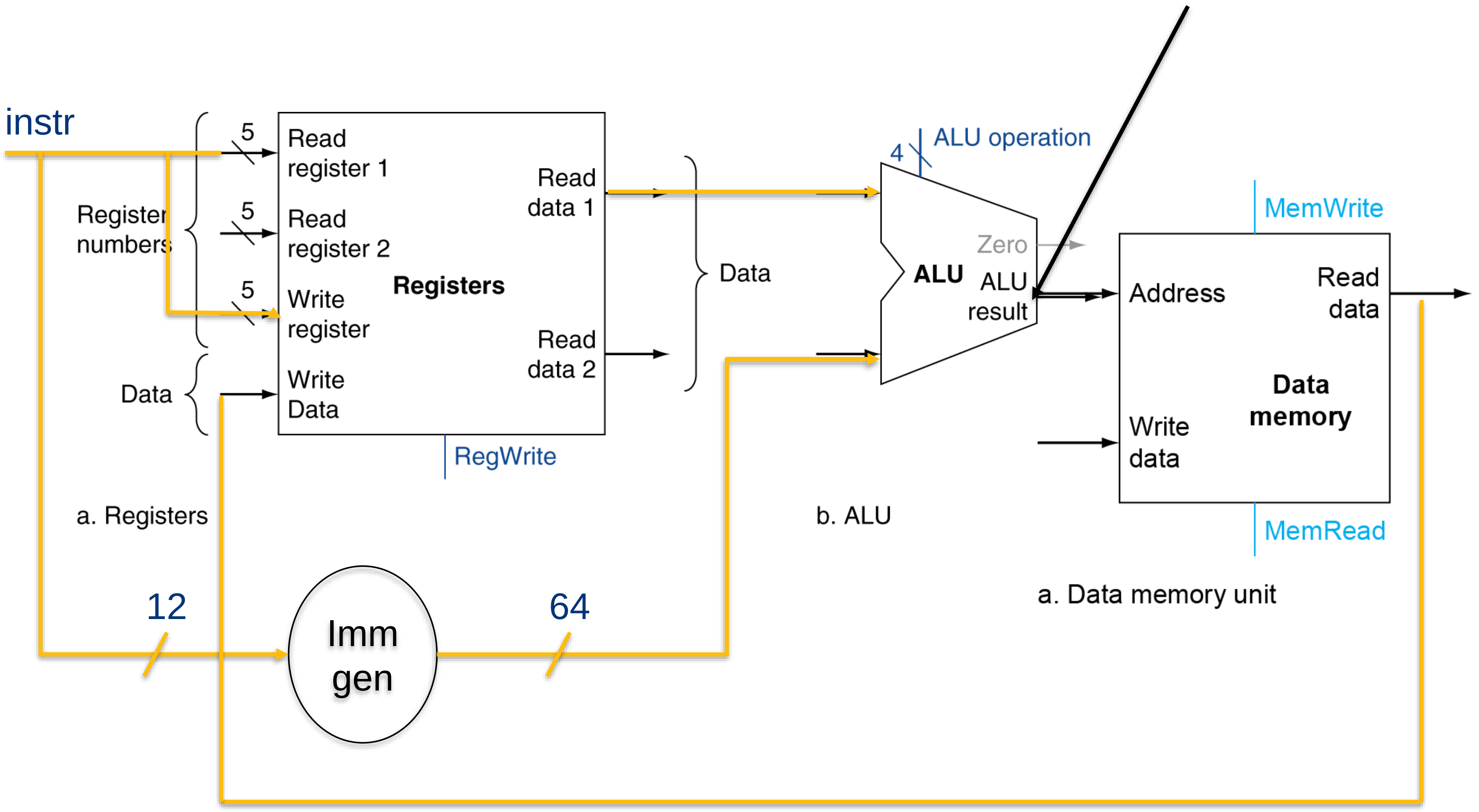


a. Data memory unit

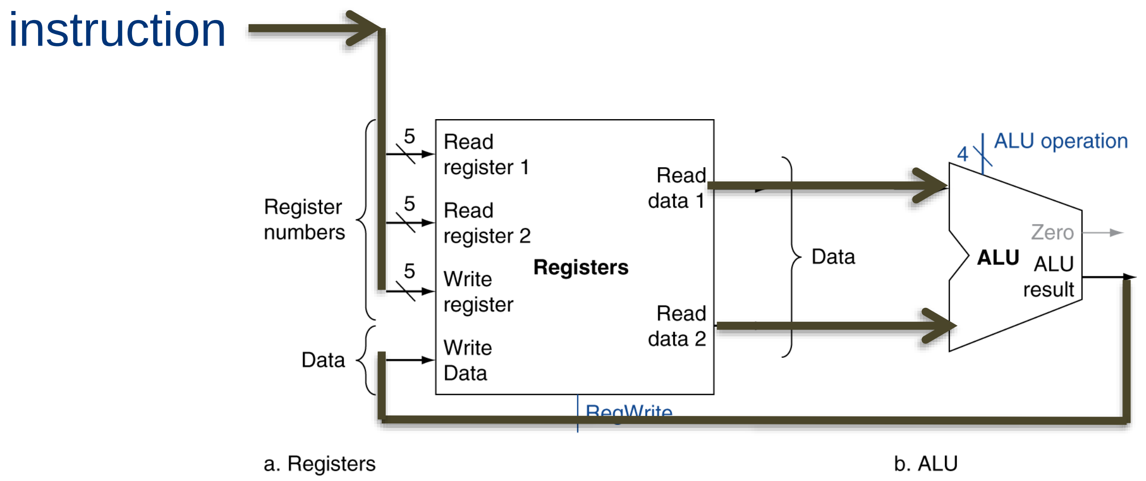


b. Immediate generatior

# Looking at Load



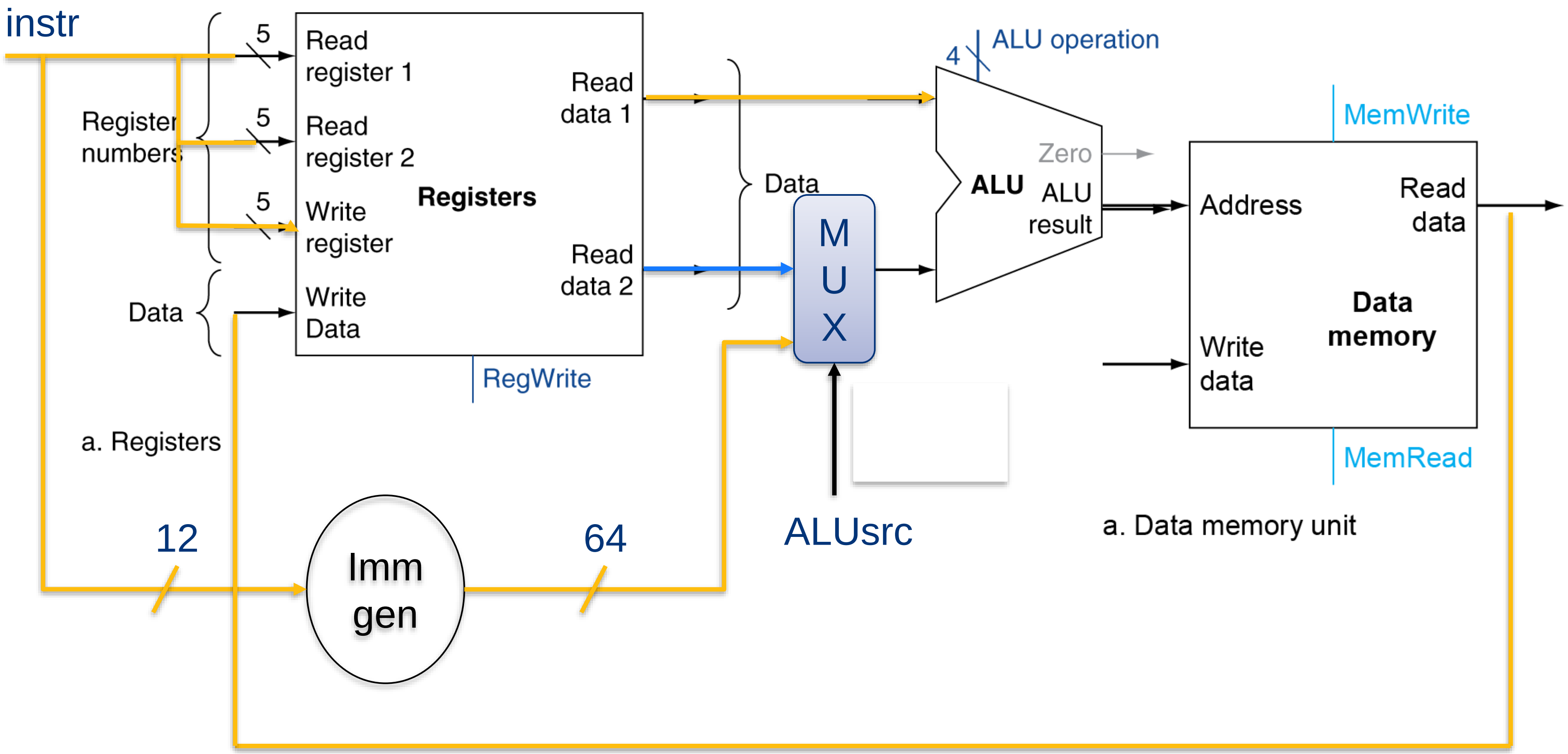
# Observe the similarity with R-type



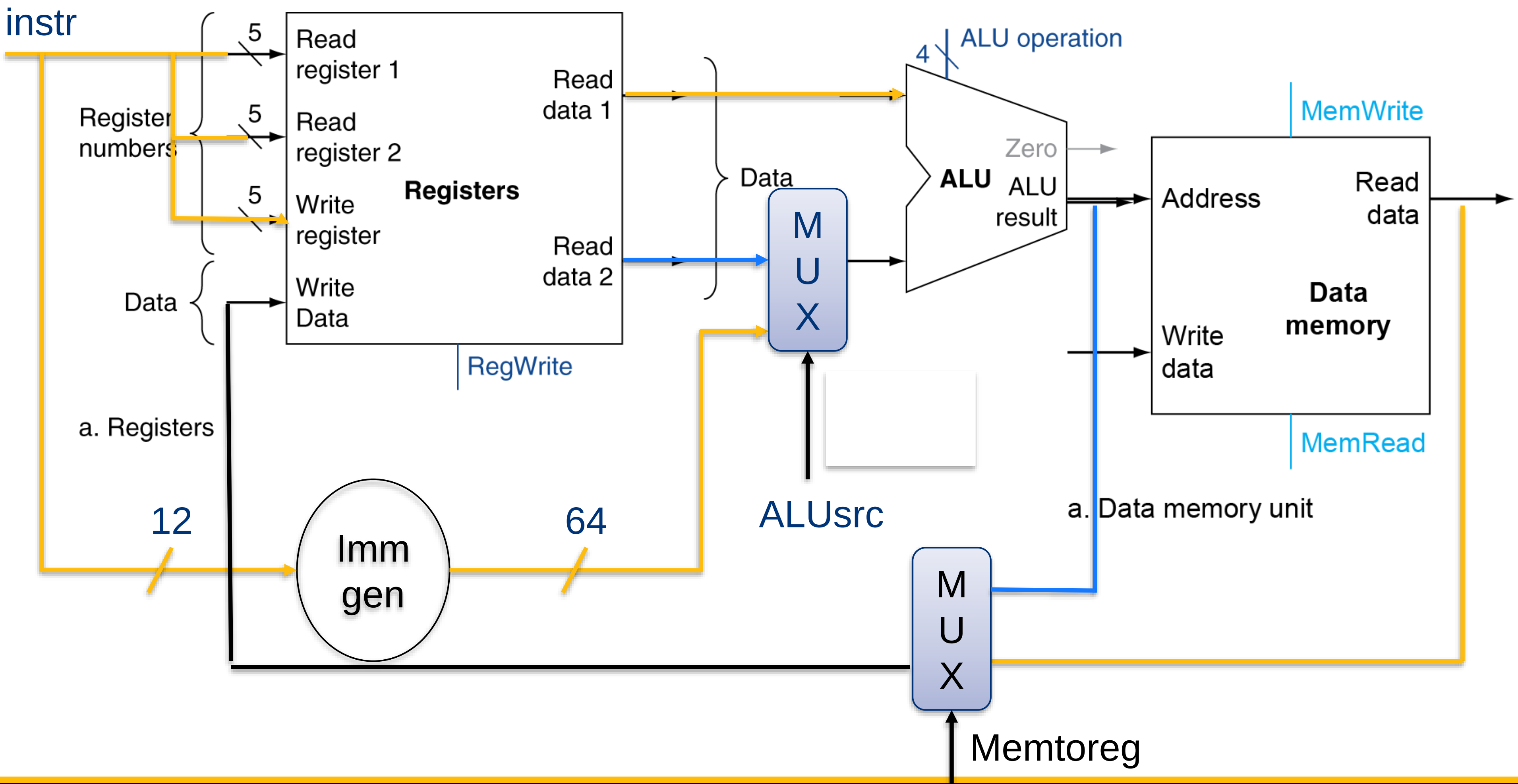
Observe Read data 2 goes to the second input of the ALU

Output of ALU goes back to the register

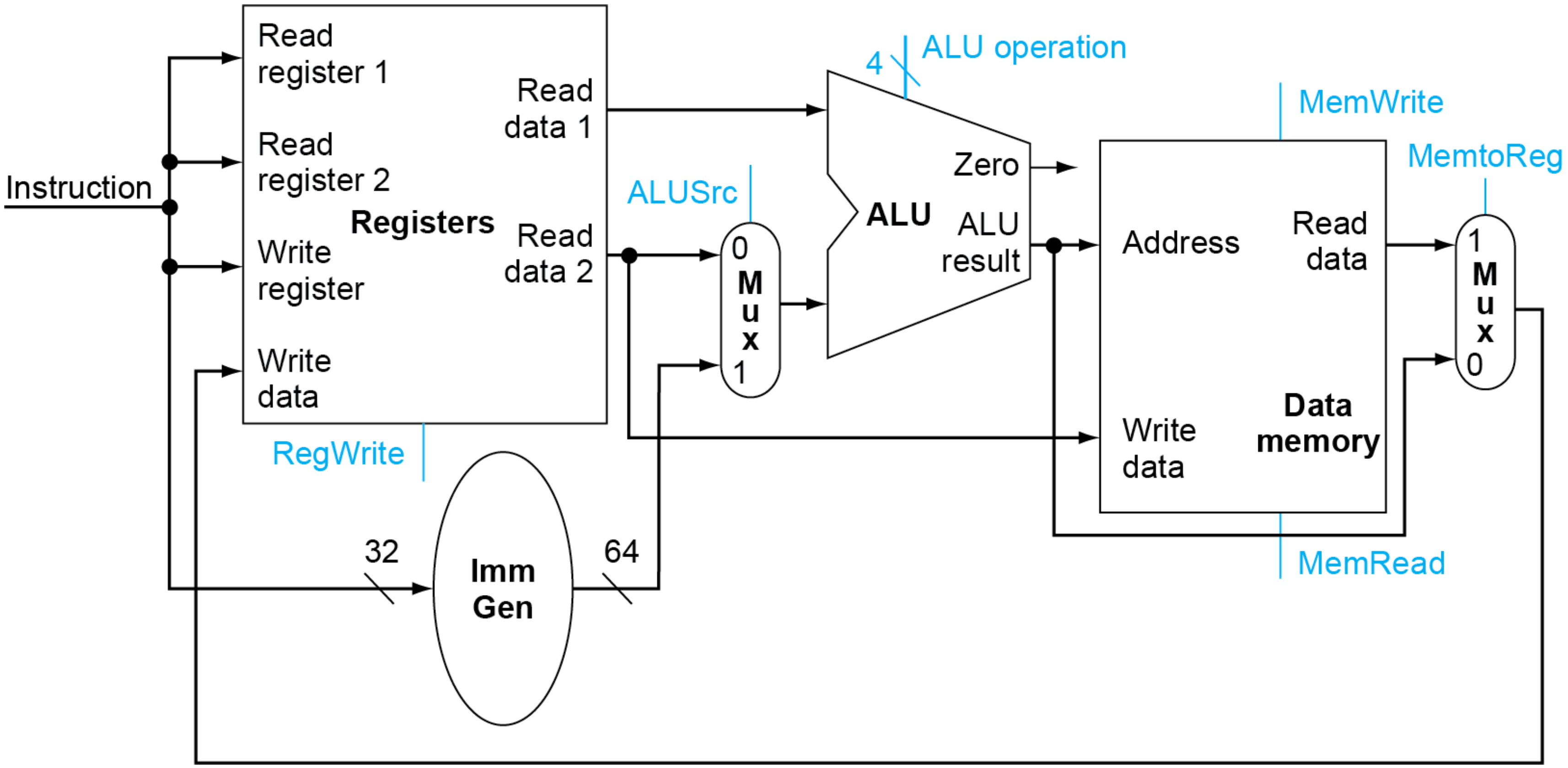
# Looking at Load



# Looking at Load



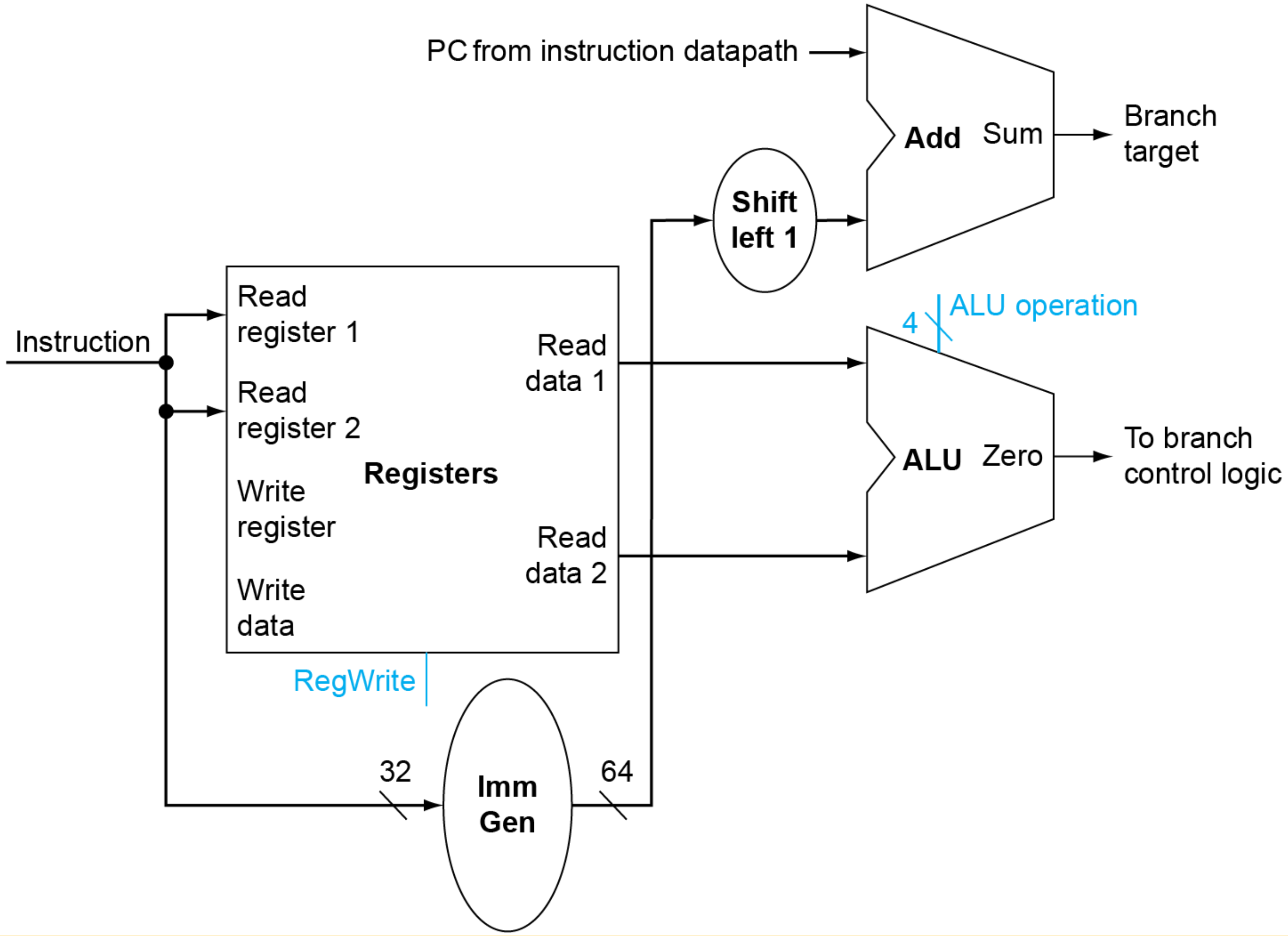
# R-Type/Load/Store Datapath



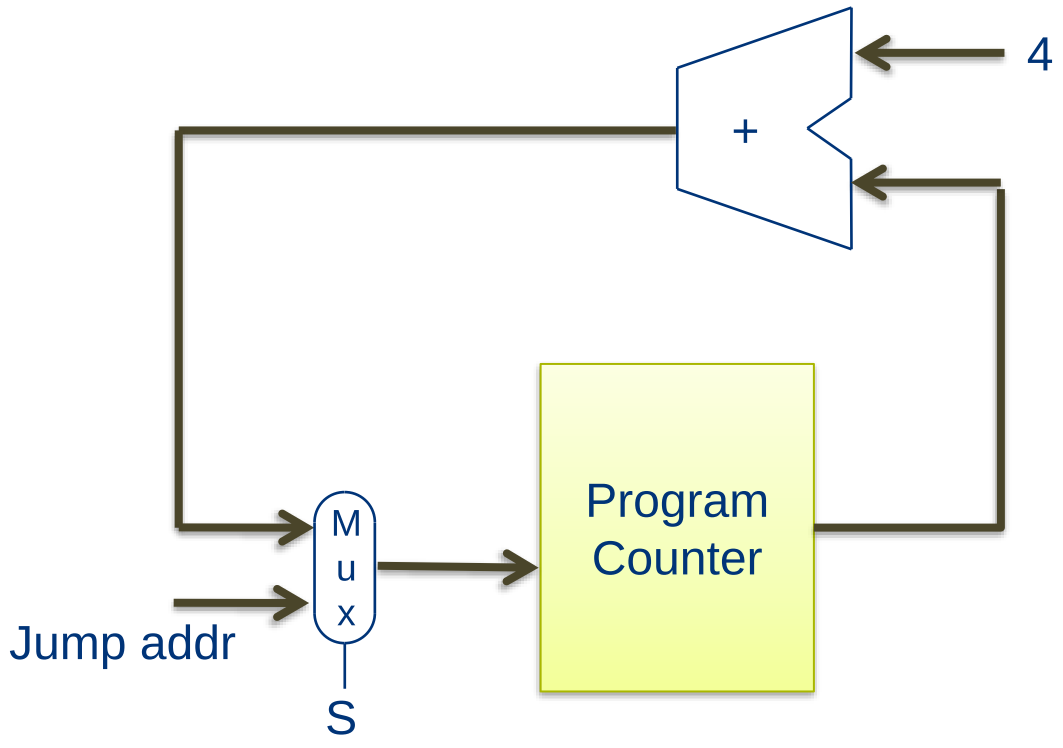
## Branch Instructions

- **Read register operands**
- **Compare operands**
  - Use ALU, subtract and check Zero output
- **Calculate target address**
  - Sign-extend displacement
  - Shift left 1 place (halfword displacement)
  - Add to PC value

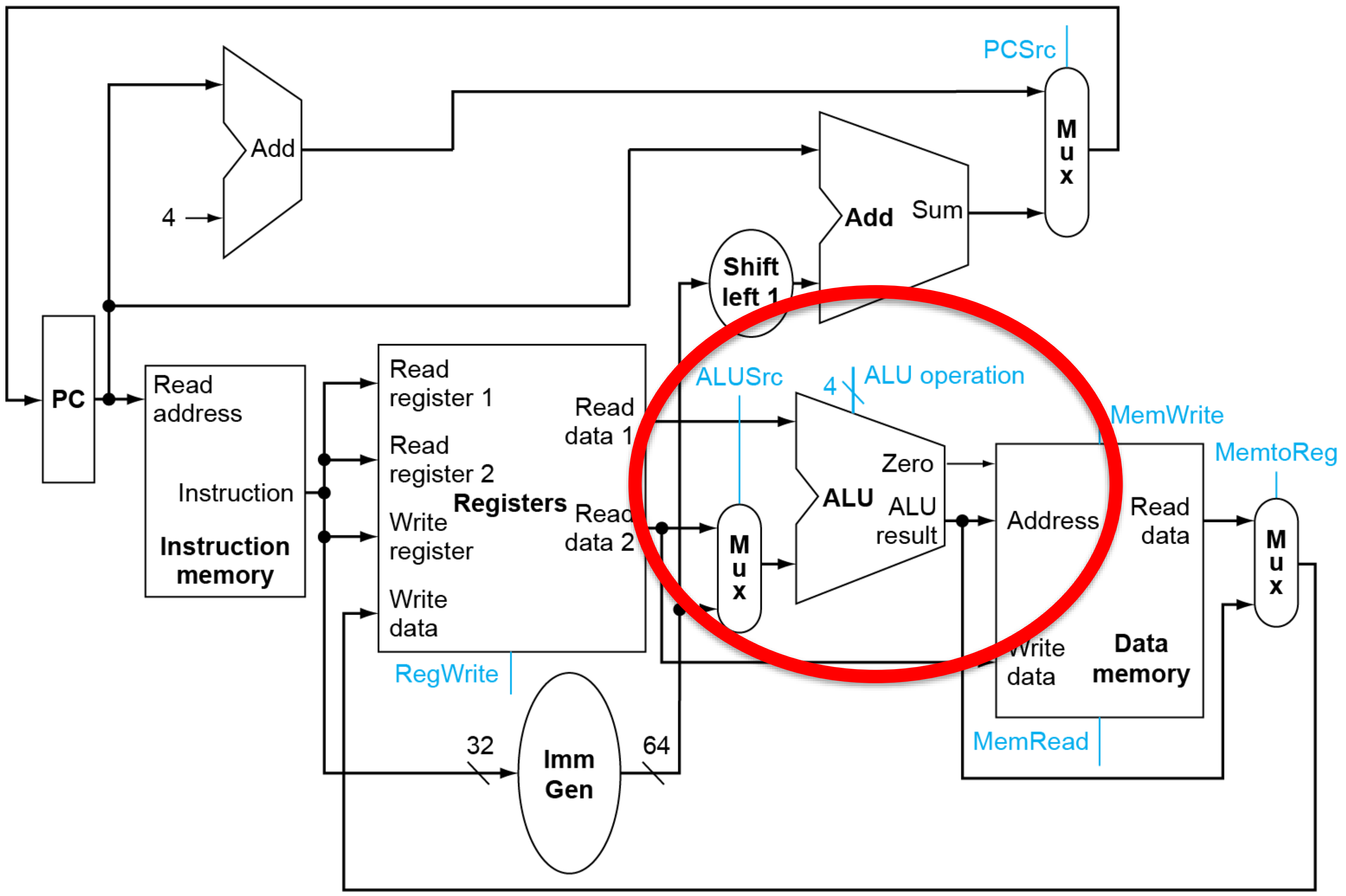
# Branch Instructions



## Remember PC?



# Full Datapath



## ALU Control

- **We build the input datapath to the ALU**
  - Basically using multiplexers

- **ALU themselves can perform lot more than addition**
  - This is controlled using 4 bits “ALU Operation”

## ALU Control

- **ALU used for**
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on opcode

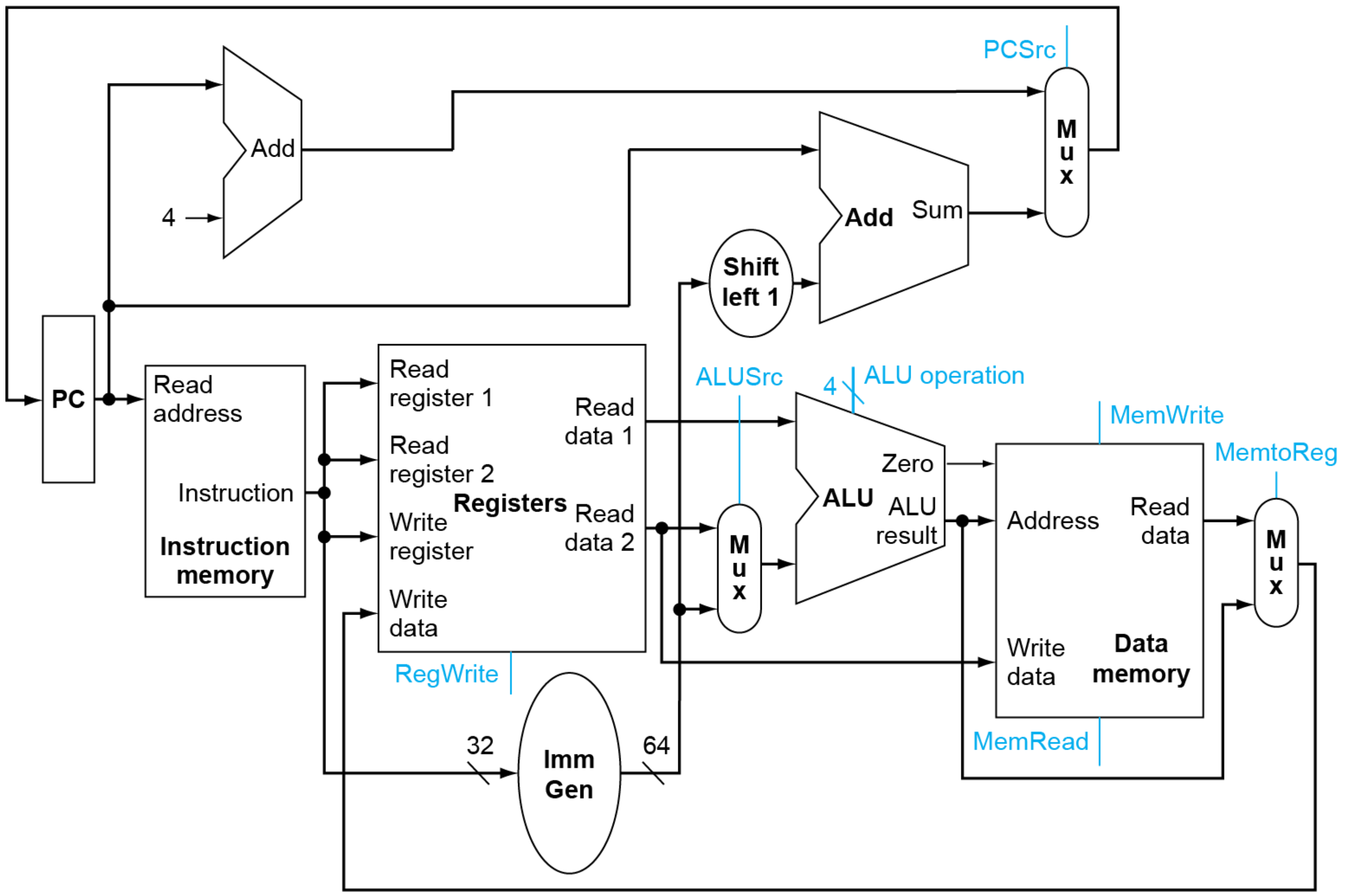
| ALU control | Function |
|---|---|
| 0000 | AND |
| 0001 | OR |
| 0010 | add |
| 0110 | subtract |

# ALU Control

- **Assume 2-bit ALUOp derived from opcode**
  - Combinational logic derives ALU control

| opcode | ALUOp | Operation | Opcode field | ALU function | ALU control |
|---|---|---|---|---|---|
| ld | 00 | load register | XXXXXXXXXXX | add | 0010 |
| sd | 00 | store register | XXXXXXXXXXX | add | 0010 |
| beq | 01 | branch on equal | XXXXXXXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| | | subtract | 100010 | subtract | 0110 |
| | | AND | 100100 | AND | 0000 |
| | | OR | 100101 | OR | 0001 |

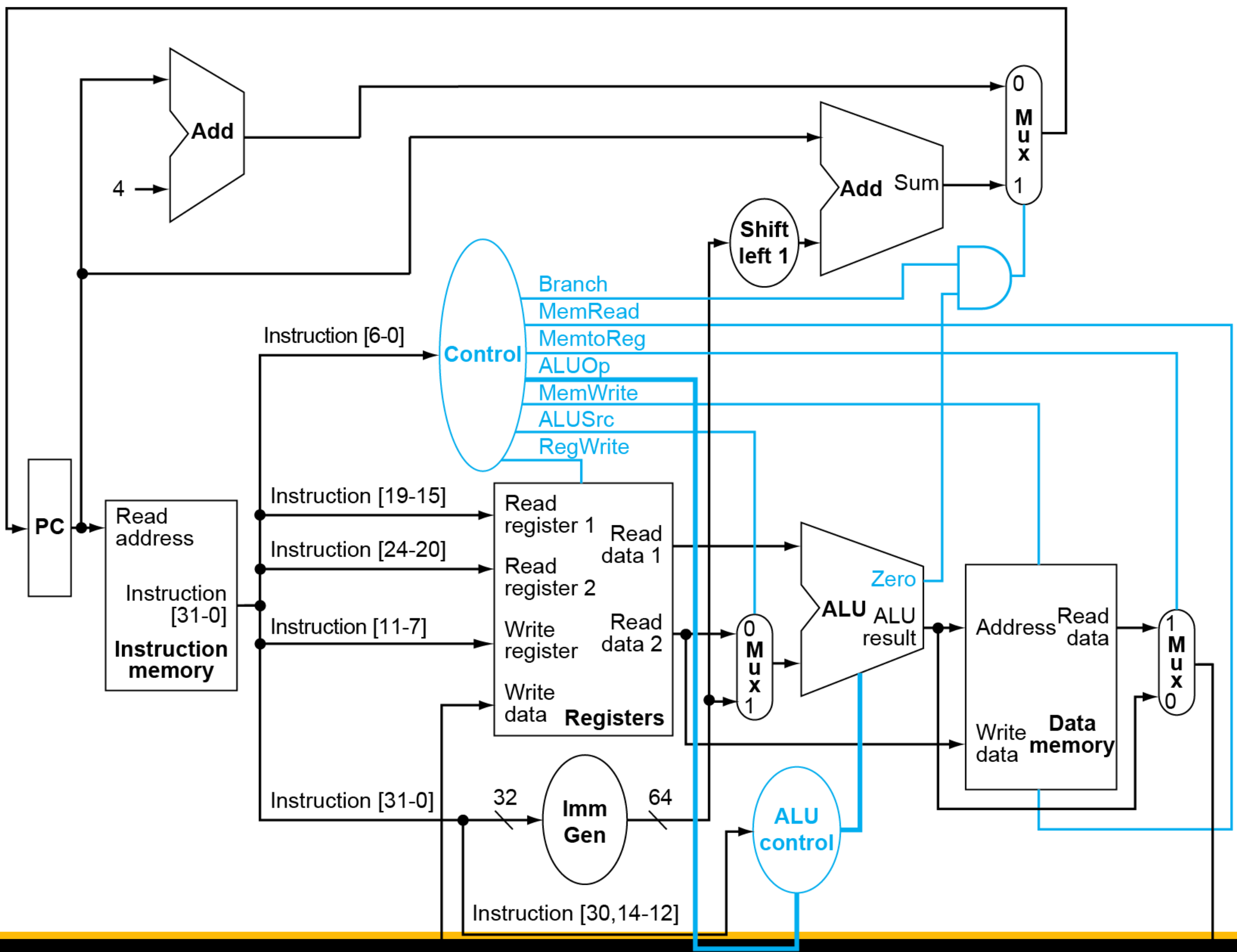
# The Main Control Unit



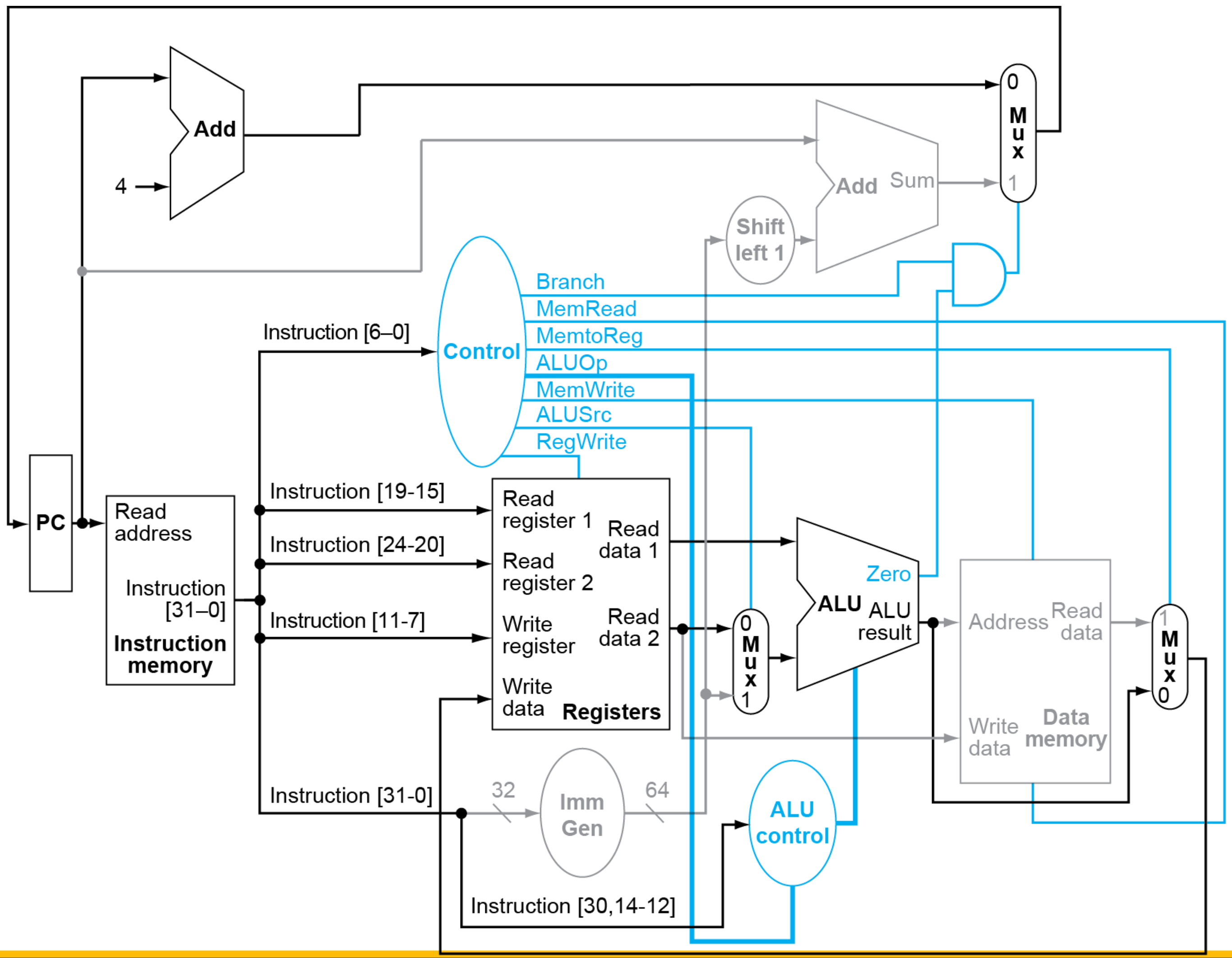
## The Main Control Unit

- **They control the Multiplexers and functional units**
  - Based on the opcode

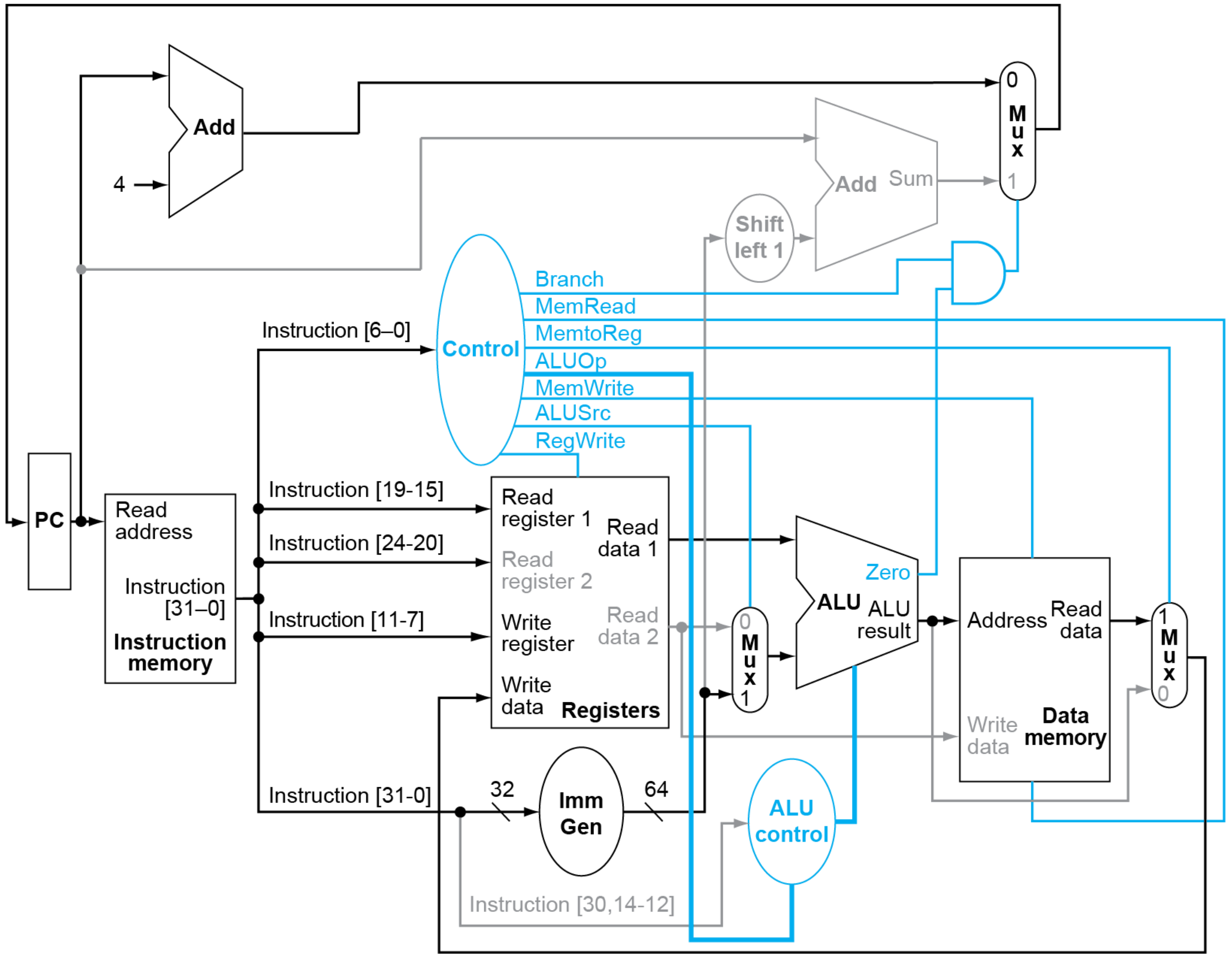
# Datapath with control



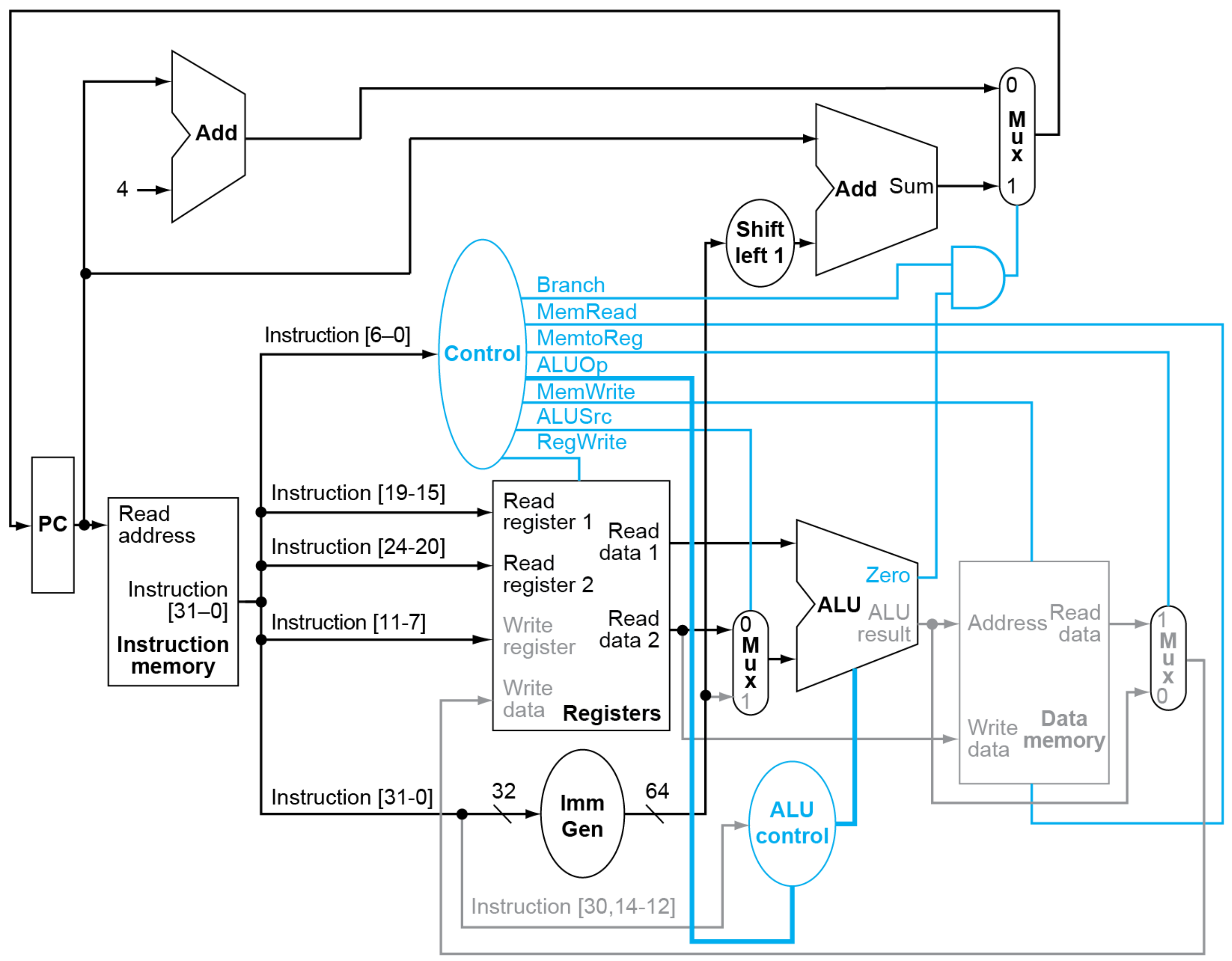
# R-Type Instruction



# Load Instruction



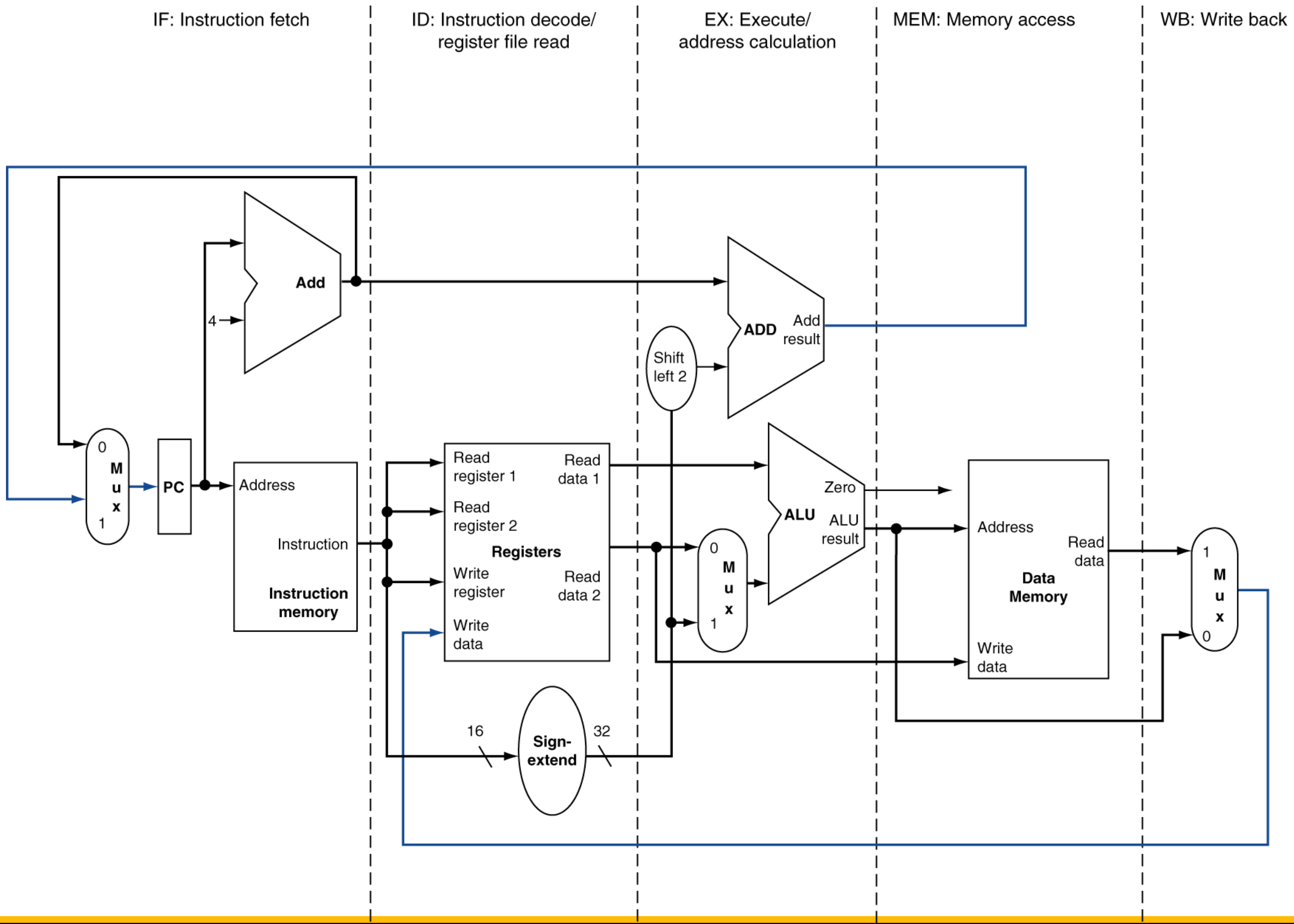
# BEQ Instruction



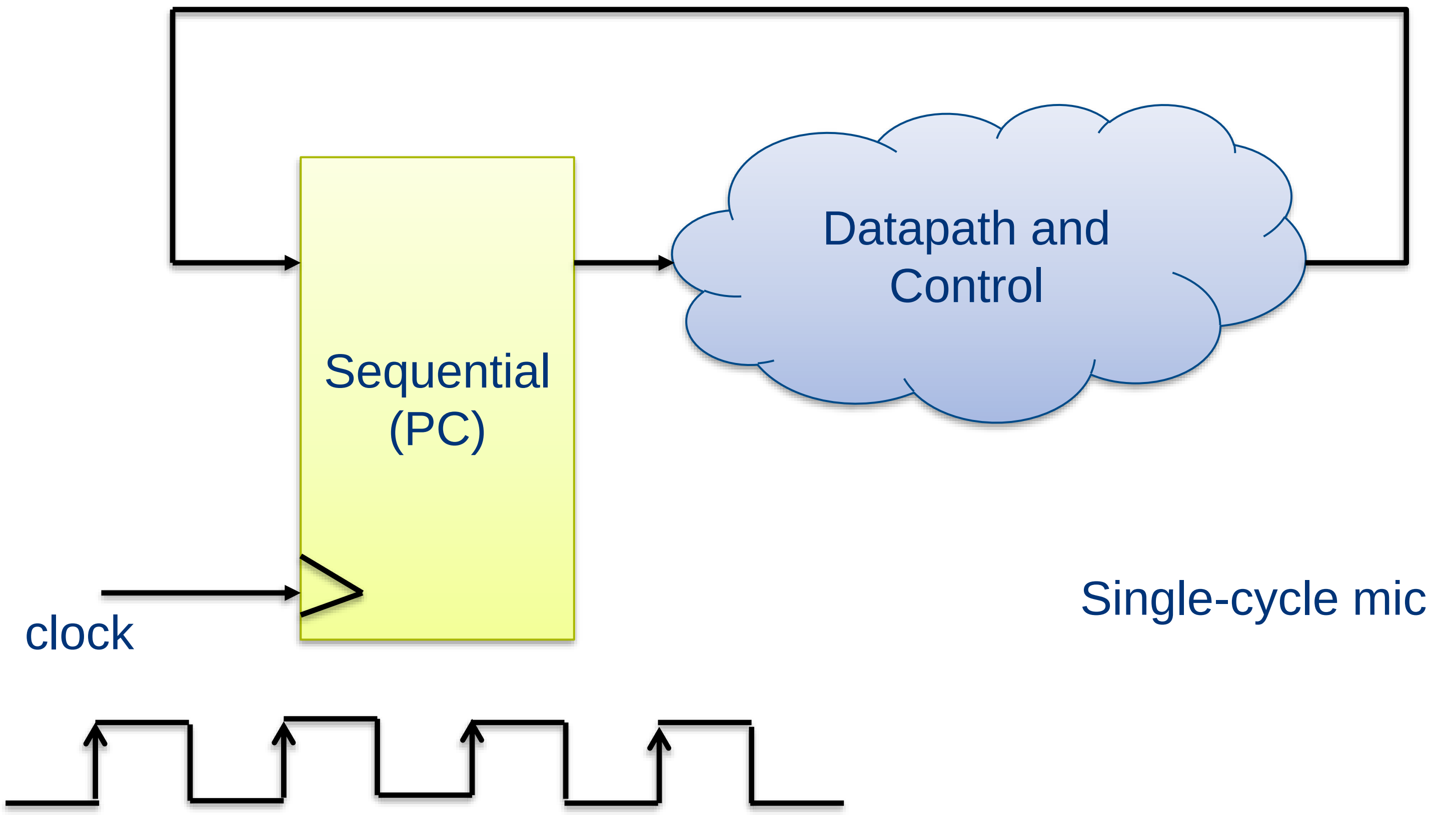
## Estimating the Datapath Delay

- **Five stages**
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

# Five Processing Stages



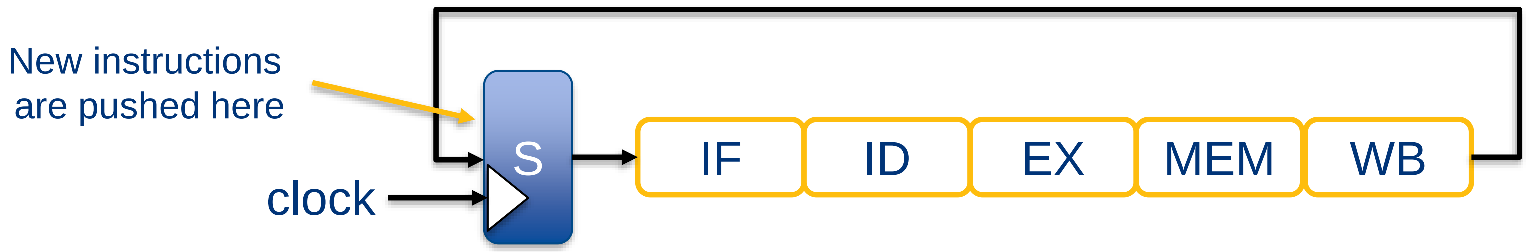
# Abstracting the Datapath



Single-cycle microarchitecture

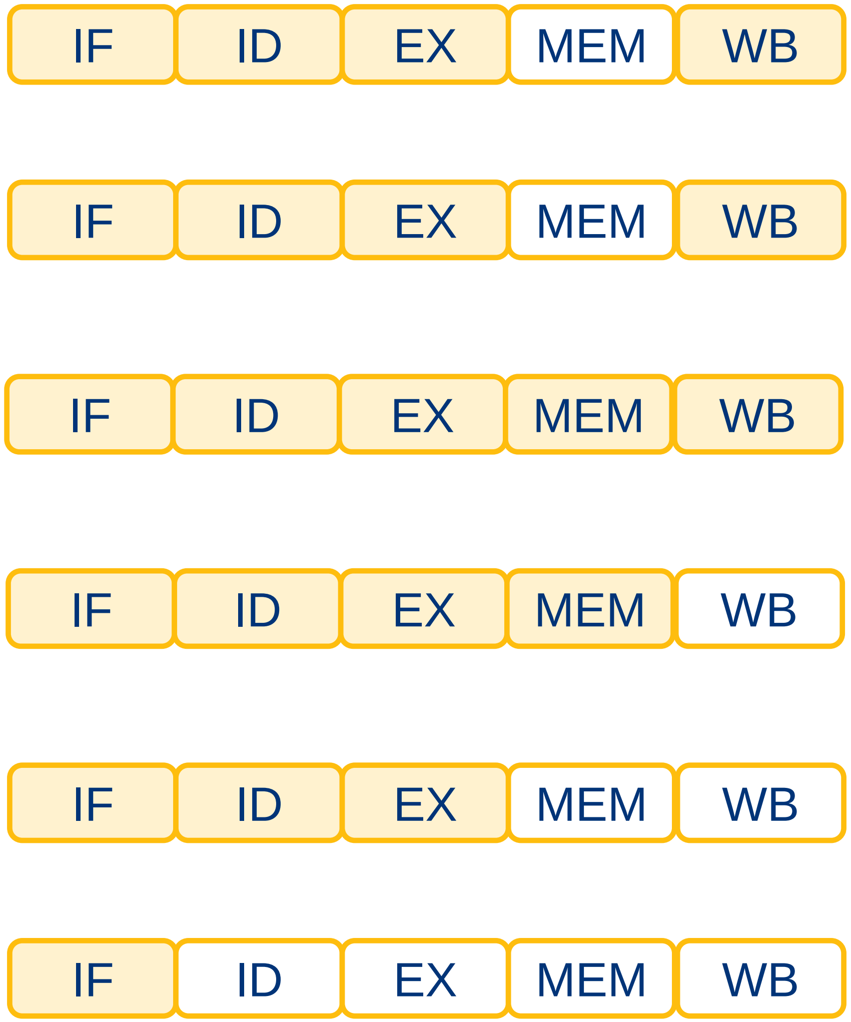
New instructions are fetched and executed starting at the positive edge
Each instruction must be completed within the clock period

## Abstracting the Datapath



# RISC-V Stages

- **R-type:**
- **I-type:**
- **Load:**
- **Store:**
- **Branch:**
- **Jump:**



## Single Cycle Microarchitecture

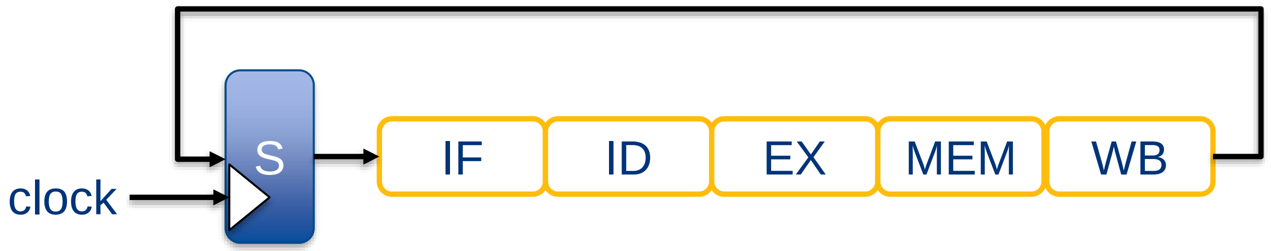
- **Design Optimization**
  - Memory units delay (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

| steps | IF | ID | EX | MEM | WB | Delay |
|---|---|---|---|---|---|---|
| resources | mem | RF | ALU | mem | RF | |
| R-type | 200 | 50 | 100 | | 50 | 400 |
| I-type | 200 | 50 | 100 | | 50 | 400 |
| LW | 200 | 50 | 100 | 200 | 50 | 600 |
| SW | 200 | 50 | 100 | 200 | | 550 |
| Branch | 200 | 50 | 100 | | | 350 |
| Jump | 200 | | | | | 200 |

## Single Cycle Microarchitecture

- **What is the minimum clock cycle time needed?**
  - 600ps

| steps | IF | ID | EX | MEM | WB | Delay |
|---|---|---|---|---|---|---|
| resources | mem | RF | ALU | mem | RF | |
| R-type | 200 | 50 | 100 | | 50 | 400 |
| I-type | 200 | 50 | 100 | | 50 | 400 |
| LW | 200 | 50 | 100 | 200 | 50 | 600 |
| SW | 200 | 50 | 100 | 200 | | 550 |
| Branch | 200 | 50 | 100 | | | 350 |
| Jump | 200 | | | | | 200 |

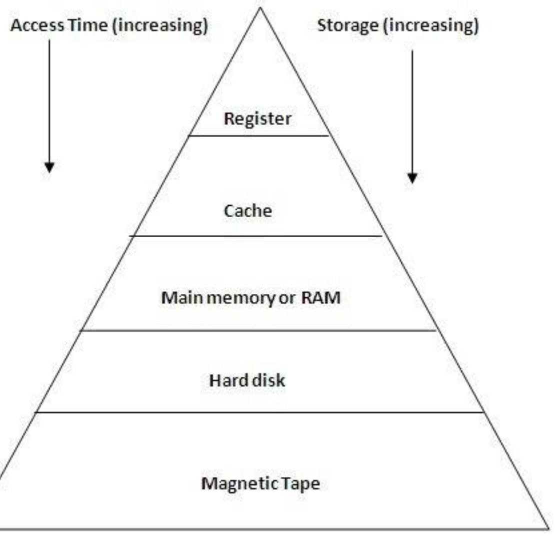


## Pitfalls

- **Memory access time can change**
- **Multiple memory access instructions**
- **Instructions requiring repetitions of some of the units**
- **Suggested readings:**
  - Johnson et al. “Using Multiple Memory Access Instructions for Reducing Code Size” in Compiler Construction 2004

# Variable Memory Access Time

- **Thus far we have assumed that data is always available in a monolithic memory**
  - Processors implement "memory hierarchy"
  - Depending on where is your data the access time can change



# Multiple Memory Access Instructions

- **Remember addressing modes?**
  - Absolute LW rt, 1000

    use 1000 as address
  - Register Indirect: LW rt, ($r_{base}$)

    use GPR[$r_{base}$] as address
  - Displaced or base: LW rt, offset($r_{base}$)

    use offset+GPR[$r_{base}$] as address
  - Indexed: LW rt, ($r_{base}$, $r_{index}$)

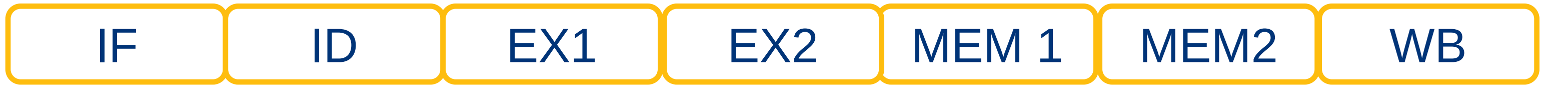
    use GPR[$r_{base}$]+GPR[$r_{index}$] as address
  - Memory Indirect LW rt (($r_{base}$))

    use value at M[ GPR[ $r_{base}$ ] ] as address
  - Auto increment/decrement LW Rt, ($r_{base}$)

    use GRP[$r_{base}$] as address, but increment or decrement GPR[$r_{base}$] each time

## Single Cycle Uarch: Design Optimization

- **Thus far we have considered simple instructions**
  - In x86, many complex instructions may require multiple execution and memory units
    - True even for MIPS and ARM



  - Memory units delay (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

  - Instruction delay = 200 + 50 + 100 + 100 + 200 + 200 + 50 = 900ps

# Single Cycle Uarch: Design Optimization

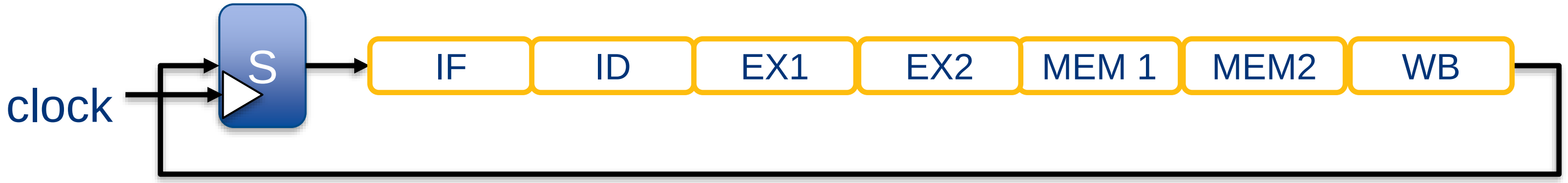
- **We have the scenario (common instructions)**

| steps | IF | ID | EX | MEM | WB | Delay |
|---|---|---|---|---|---|---|
| resources | mem | RF | ALU | mem | RF | |
| R-type | 200 | 50 | 100 | | 50 | 400 |
| I-type | 200 | 50 | 100 | | 50 | 400 |
| LW | 200 | 50 | 100 | 200 | 50 | 600 |
| SW | 200 | 50 | 100 | 200 | | 550 |
| Branch | 200 | 50 | 100 | | | 350 |
| Jump | 200 | | | | | 200 |

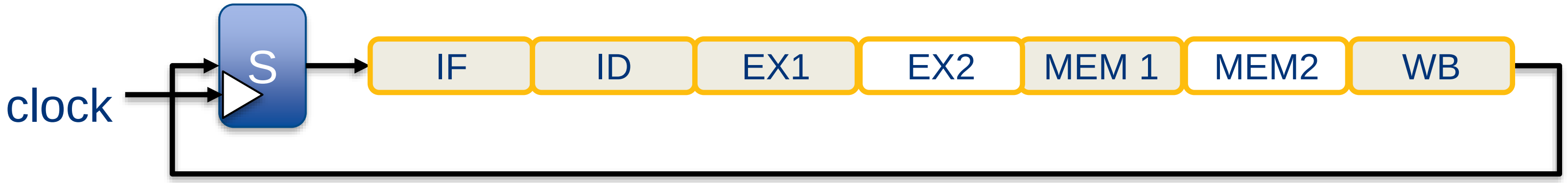
- **Some special instructions requiring 900ps**
- **Question: should we optimize for the common case (600ps) or the worst case (900ps)?**
  - Worst case. Why?
  - Otherwise, we will have design rule violation for the special cases
    - Clock is 600ps and delay is 900ps

## Single Cycle Uarch: Design Optimization

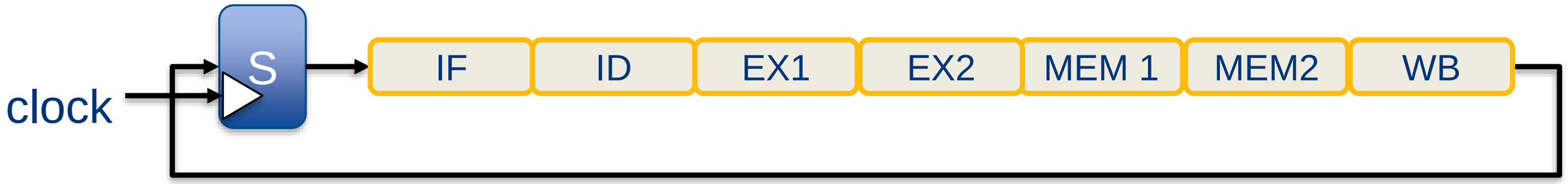
- For the example above the single-cycle microarchitecture looks like



- Common case (with 5 stages)



- Special cases (with 2 EX and 2 MEM units)



# Single Cycle Microarchitecture

- **Performance**
  - CPI = 1
  - Clock Cycle Time = 900 ps (worst case)
  - Execution time of 100 billion instructions
    - = {# of instructions} x {Average CPI} x {clock cycle time}
    - = {100 x $10^{9}$} x 1 x {900 x $10^{-12}$}
    - = 90 seconds
  - Average latency of each instruction = 1 cycle = 900 ps
  - Average instruction throughput = 1 / (1 x 900 x $10^{-12}$) = 1.1 billion instructions per second
  - IPC = 1 / CPI = 1

# Performance Comparison

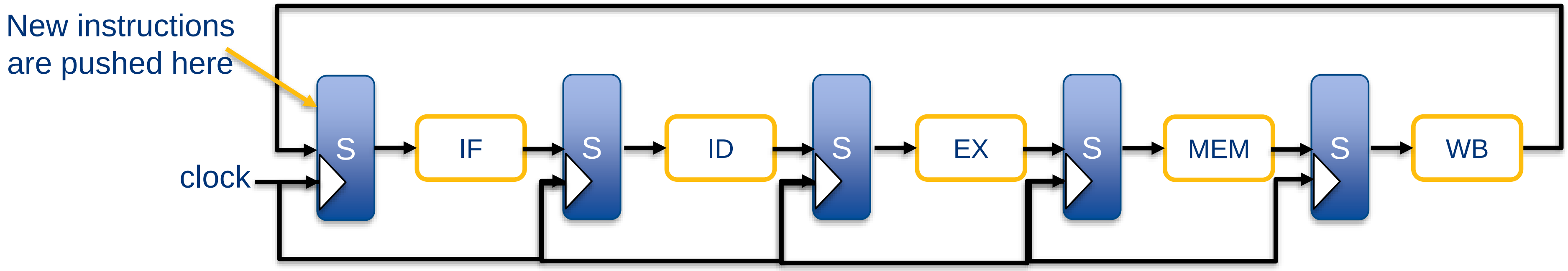
| | Single cycle | | |
|---|---|---|---|
| CPI | 1 | | |
| Clock cycle time | 900 ps | | |
| Execution time of 100 billion instructions | 90 s | | |
| Average latency of each instruction | 900 ps | | |
| Average instruction throughput | 1.1 bips | | |
| IPC | 1 | | |

## Single Cycle Microarchitecture: Concluding Remarks

- **Contrived**
  - All instructions run as slow as the slowest instruction
- **Inefficient**
  - All instructions run as slow as the slowest instruction
  - Must provide worst-case combinational resources in parallel as required by any instruction
  - Need to replicate a resource if it is needed more than once by an instruction during different parts of the instruction processing cycle
- **Not necessarily the simplest way to implement an ISA**
- **Not easy to optimize/improve performance**
  - Optimizing the common case does not work (e.g. common instructions)
  - Need to optimize the worst case all the time

# Multi-cycle Microarchitecture

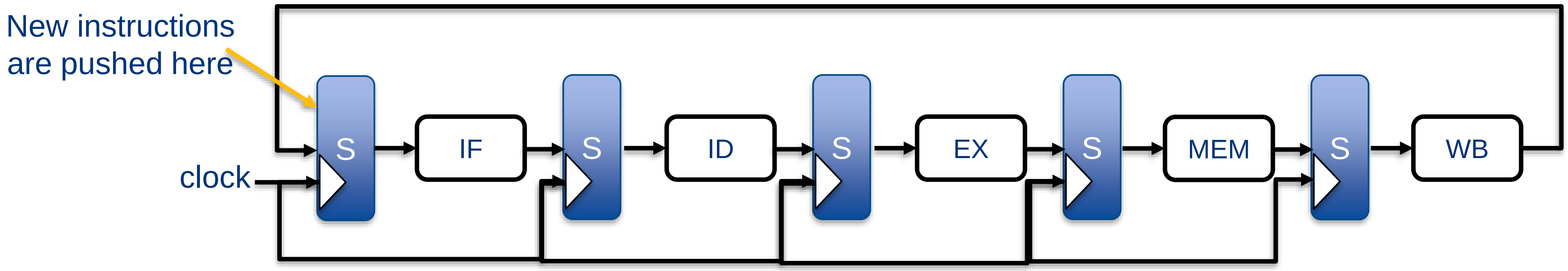
- **Idea: Start with the 5 stages and insert sequential logic after each stage**



- **How does this microarchitecture mitigates the limitations of the single-cycle architecture?**
- **How does this microarchitecture handles special long instructions?**
- **What are performance metric using this microarchitecture?**

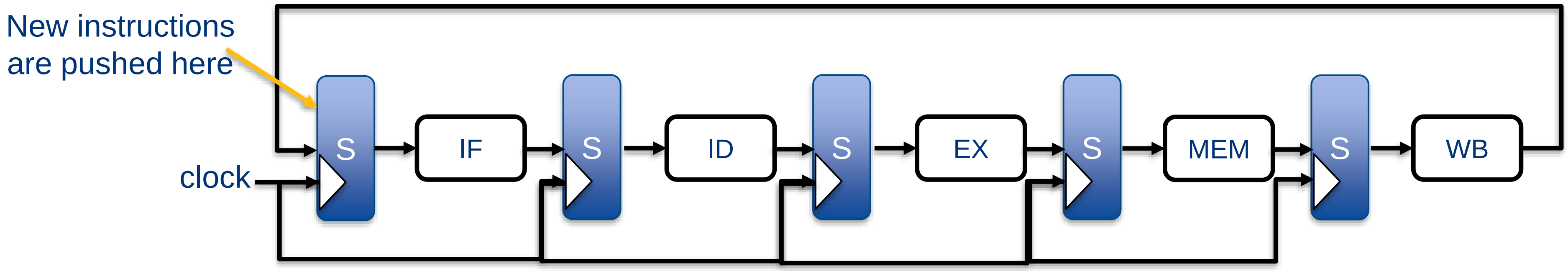
# Visualization

- Inst 1
- Inst 2
- Inst 3
- …



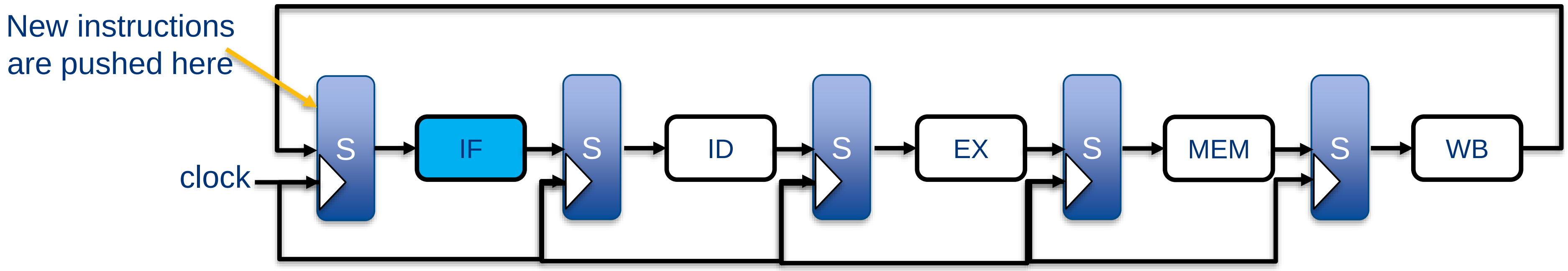
# Visualization

- Inst 1
- Inst 2
- Inst 3
- …



# Visualization

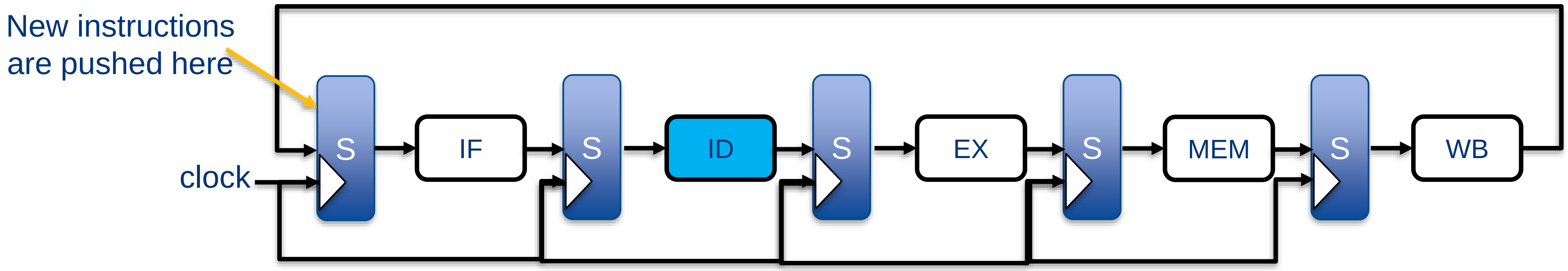
- Inst 1
- Inst 2
- Inst 3
- …



Clock Cycle 1

# Visualization

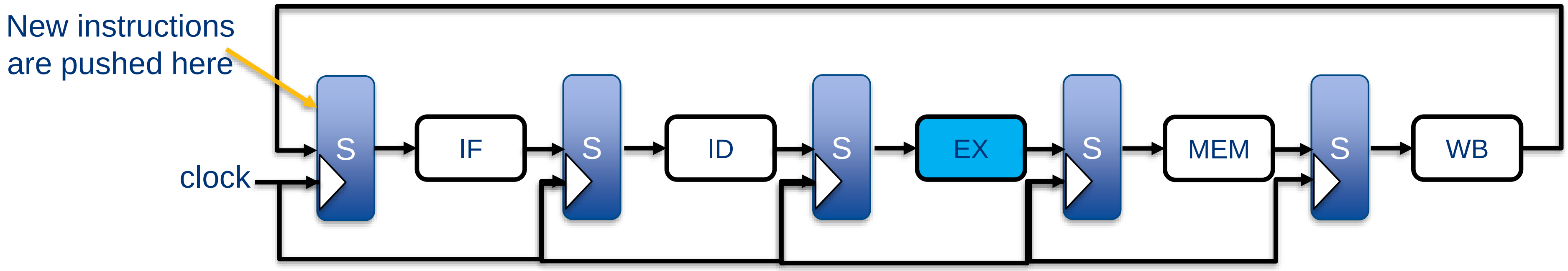
- Inst 1
- Inst 2
- Inst 3
- …



Clock Cycle 2

# Visualization

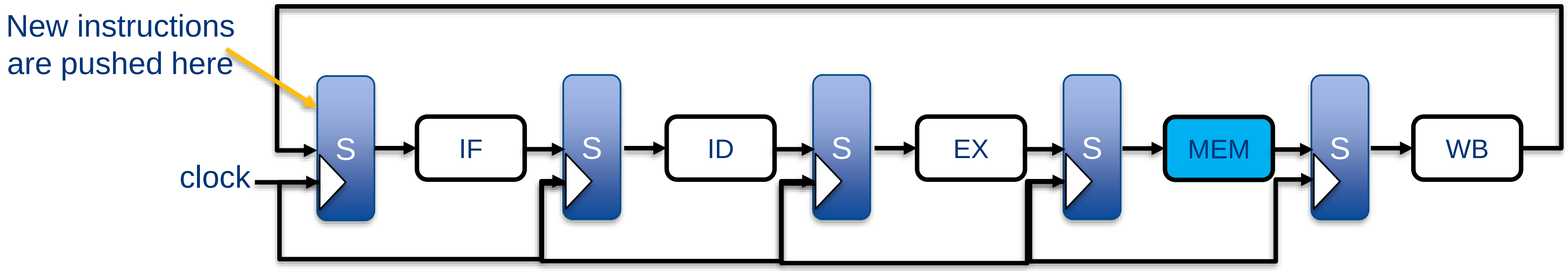
- Inst 1
- Inst 2
- Inst 3
- …



Clock Cycle 3

# Visualization

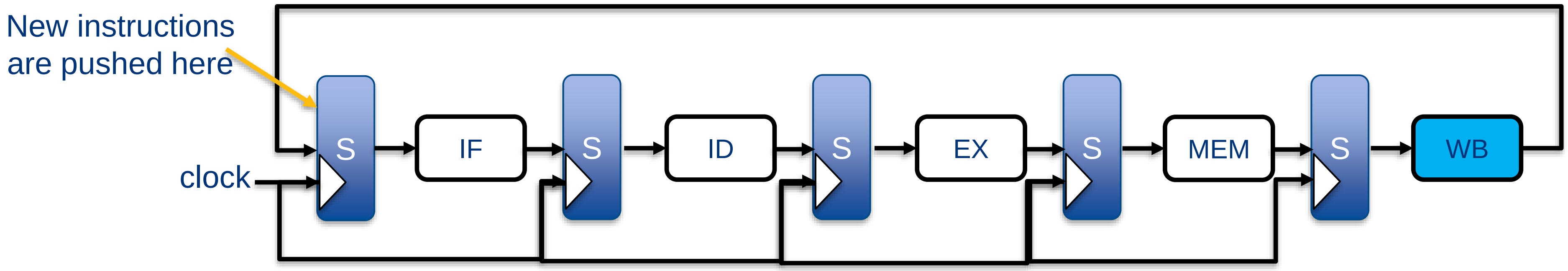
- Inst 1
- Inst 2
- Inst 3
- …



Clock Cycle 4

# Visualization

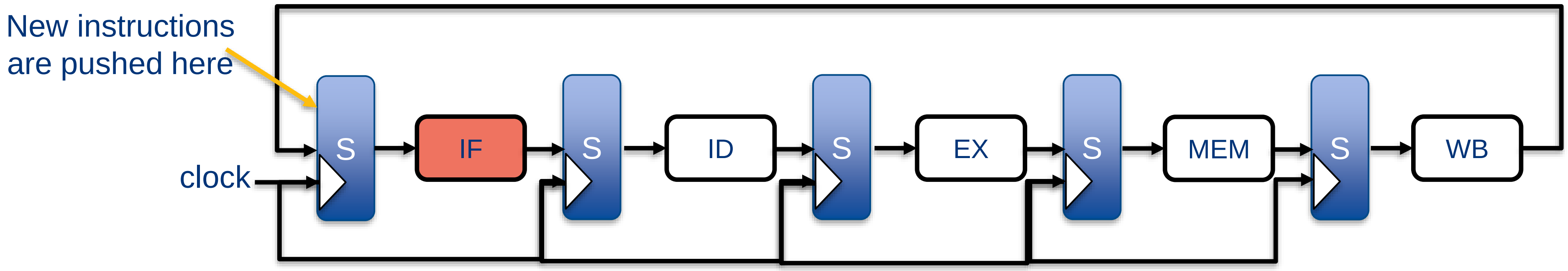
- Inst 1
- Inst 2
- Inst 3
- …



Clock Cycle 5

# Visualization

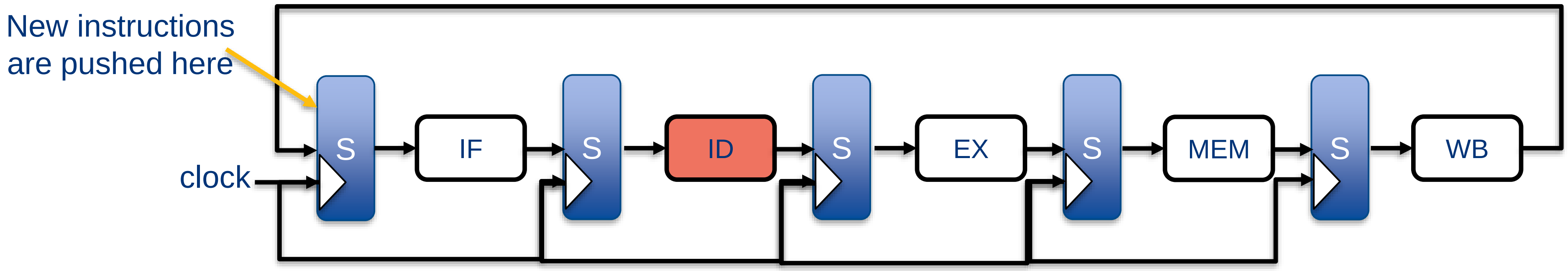
- Inst 1
- Inst 2
- Inst 3
- …



Clock Cycle 6

# Visualization

- Inst 1
- Inst 2
- Inst 3
- …



Clock Cycle 7

And so on

## Summary

| Instr | Clocks | IF | ID | EX | MEM | WB |
|---|---|---|---|---|---|---|
| 1 | 1 | X | | | | |
| | 2 | | X | | | |
| | 3 | | | X | | |
| | 4 | | | | X | |
| | 5 | | | | | X |
| 2 | 6 | X | | | | |
| | 7 | | X | | | |
| | 8 | | | X | | |
| | 9 | | | | X | |
| | 10 | | | | | X |
| 3 | 11 | X | | | | |
| | 12 | | X | | | |

# Many Observations

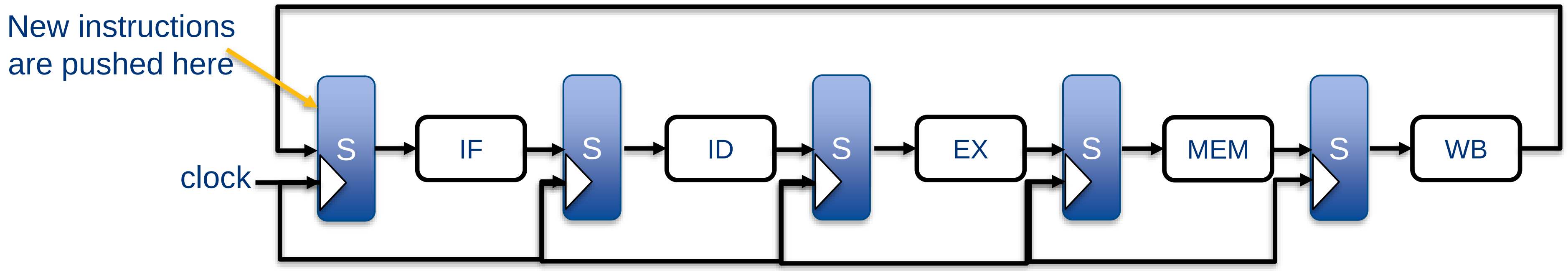
- **A new instruction starts processing only when previous instruction finishes**
- **For a given instruction, only one execution unit is occupied at any given time**
  - Under utilization of resources
  - When instruction 1 is in the decode stage, a new instruction could have used the fetch stage (pipelining, more later)
- **A single instruction now takes 5 cycles**
  - What about non standard instructions ?

    (standard = one that takes all 5 stages)

| Instr | Clocks | IF | ID | EX | MEM | WB |
|---|---|---|---|---|---|---|
| 1 | 1 | X | | | | |
| | 2 | | X | | | |
| | 3 | | | X | | |
| | 4 | | | | X | |
| | 5 | | | | | X |
| 2 | 6 | X | | | | |
| | 7 | | X | | | |
| | 8 | | | X | | |
| | 9 | | | | X | |
| | 10 | | | | | X |
| 3 | 11 | X | | | | |
| | 12 | | X | | | |

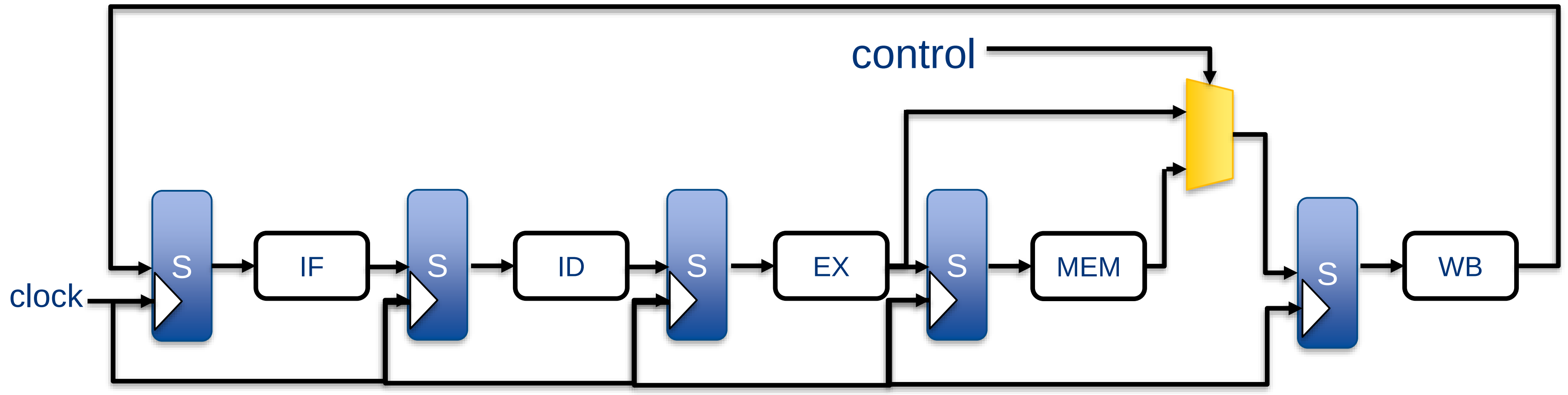
## Smaller Instructions

- **Bypass stages**
  - Example of R-type Instruction: IF, ID, EX, WB (no MEM stage needed)



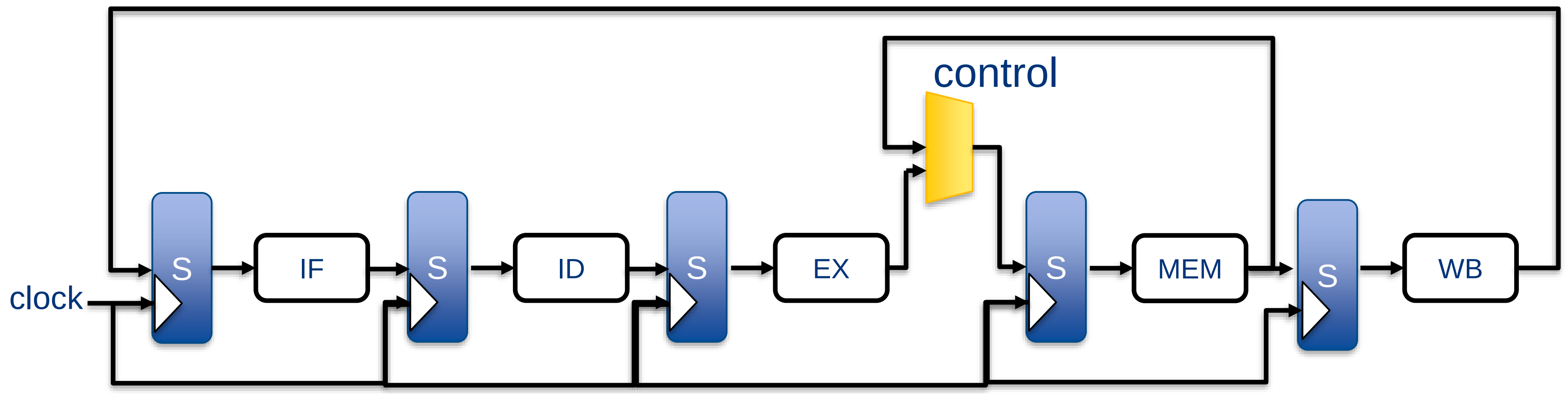
## Smaller Instructions

- **Bypass stages**
  - Example of R-type Instruction: IF, ID, EX, WB (no MEM stage needed)



# Longer Instructions

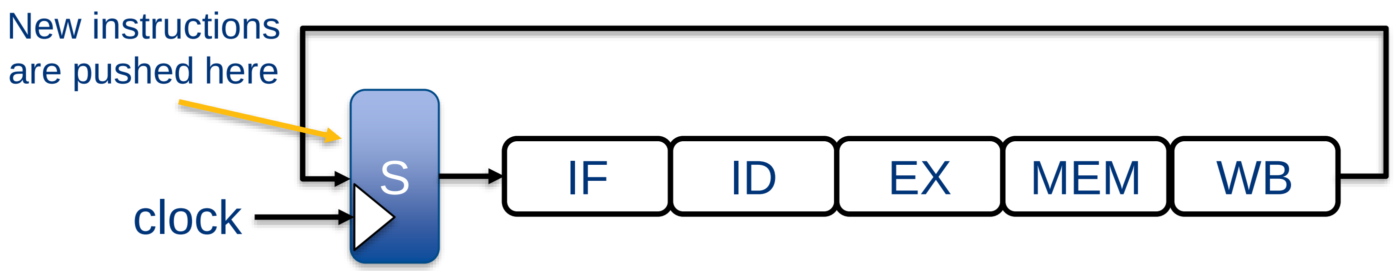
- **Example: Instruction requiring 2 MEM stages**

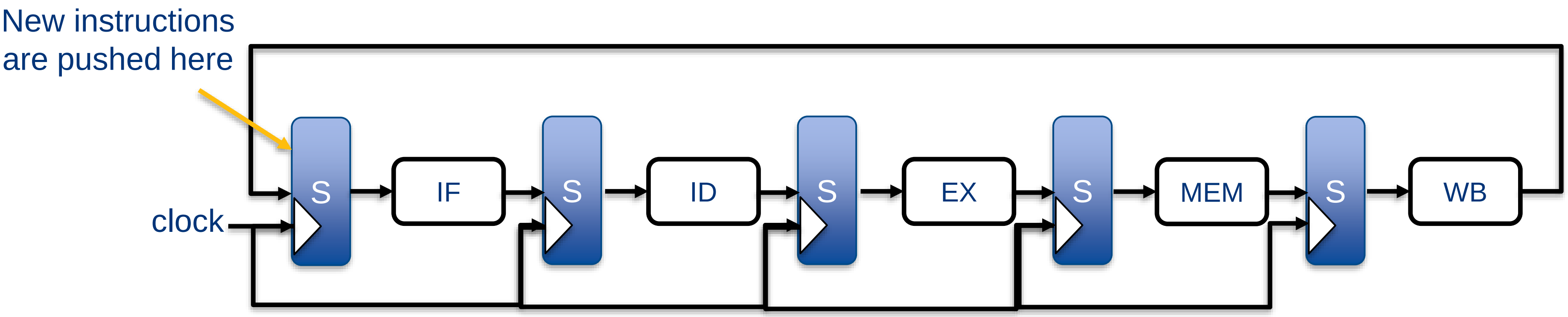


# Multi-cycle Microarchitecture

- **Singe vs Mult-Cycle**

Critical path = entire stages





Critical path = each stage

# Multi-cycle Microarchitecture

- **Critical path design**
  - Can keep reducing the critical path independently of the worst-case processing time of any instruction
- **common case design**
  - Can optimize the number of states it takes to execute “important” instructions that make up much of the execution time
- **Balanced design**
  - No need to provide more capability or resources than really needed
    - An instruction that needs resource X multiple times does not require multiple X’s to be implemented
    - Leads to more efficient hardware: Can reuse hardware components needed multiple times for an instruction

## Multi-cycle Microarchitecture: performance

- **Execution time of an instruction**
  - {CPI} x {clock cycle time}
- **Execution time of a program**
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - {# of instructions} x {Average CPI} x {clock cycle time}

- **Single cycle microarchitecture performance**
  - CPI = 1
  - Clock cycle time = long
- **Multi-cycle microarchitecture performance**
  - CPI = different for each instruction
    - Average CPI → hopefully small
  - Clock cycle time = short

## Multi-cycle Microarchitecture: performance

- **Instruction takes different number of cycles**
  - 3 cycles
    - branch
  - 4 cycles
    - R-type, store, addi
  - 5 cycles
    - load
- **CPI = Average/weighted average**
  - SPECINT2000 benchmarks
    - 25% load
    - 10% store
    - 13% branches
    - 52% R-type
  - Average CPI = (25*5 + 10*4 + 13*3 + 52*4) / (25 + 10 + 13 + 52)

    = 4.12

# Multi-cycle Microarchitecture: Performance

| steps | IF | ID | EX | MEM | WB | Delay |
|---|---|---|---|---|---|---|
| resources | mem | RF | ALU | mem | RF | |
| R-type | 200 | 50 | 100 | | 50 | 400 |
| I-type | 200 | 50 | 100 | | 50 | 400 |
| LW | 200 | 50 | 100 | 200 | 50 | 600 |
| SW | 200 | 50 | 100 | 200 | | 550 |
| Branch | 200 | 50 | 100 | | | 350 |
| Jump | 200 | | | | | 200 |

What should be the clock cycle?: 3 numbers: 200, 100, 50
- 200ps

## Multi-cycle Microarchitecture: Performance

- **CPI = 4.12**
- **Clock Cycle Time = 200ps**
- **Execution time of 100 billion instructions**
  - = {# of instructions} x {Average CPI} x {clock cycle time}
  - = {100 x $10^{9}$} x 4.12 x {200 x $10^{-12}$}
  - = 82.4 seconds
- **Average latency of each instruction = 4.12 cycles = 4.12 x 200 = 824ps**
- **Average instruction throughput = 1 / (4.12 x 200 x $10^{-12}$) = 1.21 billion instructions per second**
- **IPC = 1 / CPI = 1 / 4.12 = 0.24**

# Performance Comparison

| | Single cycle | Multi-cycle | |
|---|---|---|---|
| CPI | 1 | | |
| Clock cycle time | 900 ps | | |
| Execution time of 100 billion instructions | 90 s | | |
| Average latency of each instruction | 900 ps | | |
| Average instruction throughput | 1.1 bips | | |

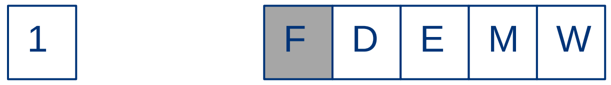
# Performance Comparison

| | Single cycle | Multi-cycle | |
|---|---|---|---|
| CPI | 1 | 4.12 | |
| Clock cycle time | 900ps | 200ps | |
| Execution time of 100 billion instructions | 90s | 82.4s | |
| Average latency of each instruction | 900ps | 824ps | |
| Average instruction throughput | 1.1 bips | 1.21 bips | |

How can we improve these metrices?

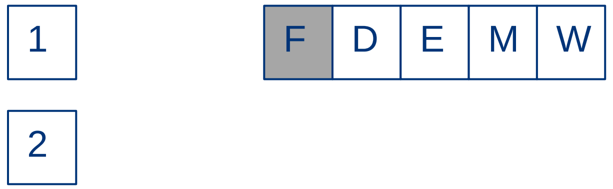
# Representations

clock State of the microarchitecture

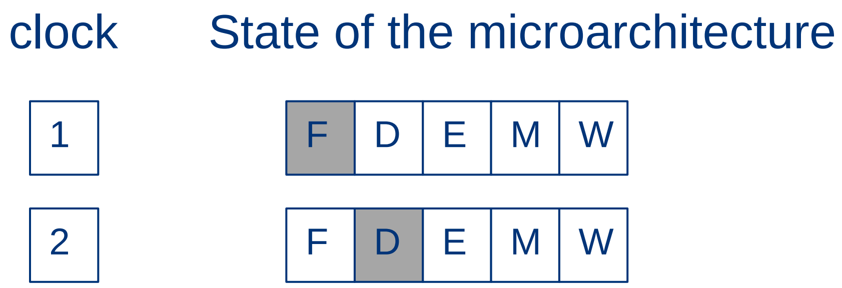


# Representations

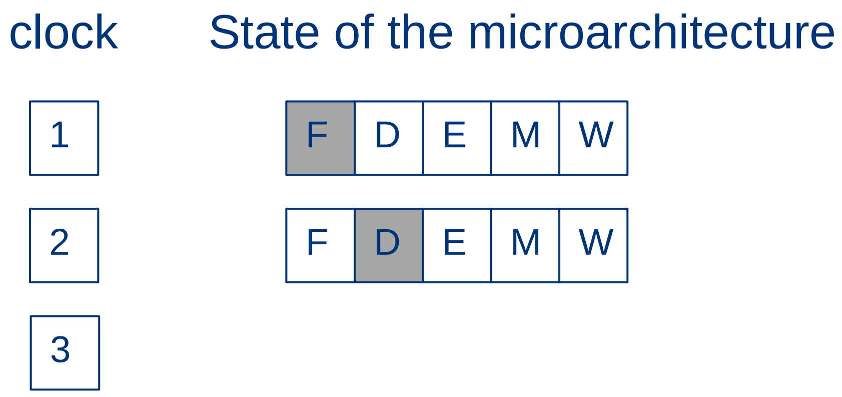
clock        State of the microarchitecture



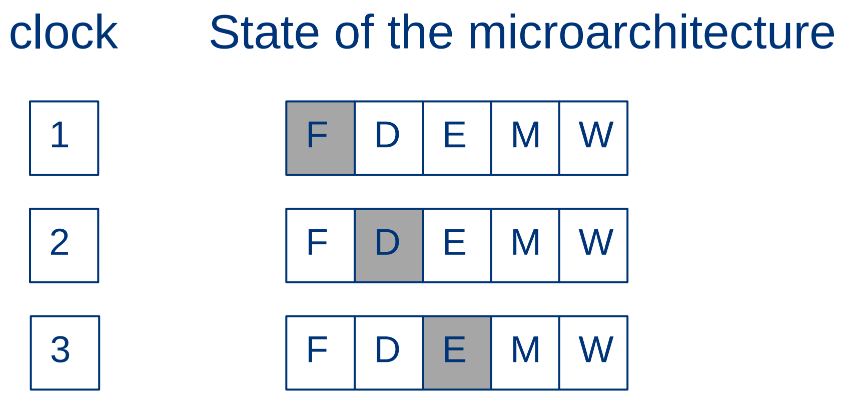
# Representations



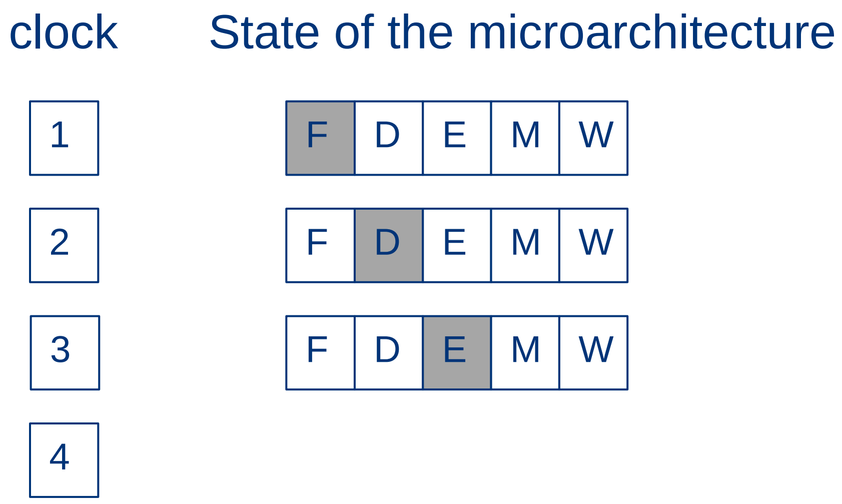
## Representations



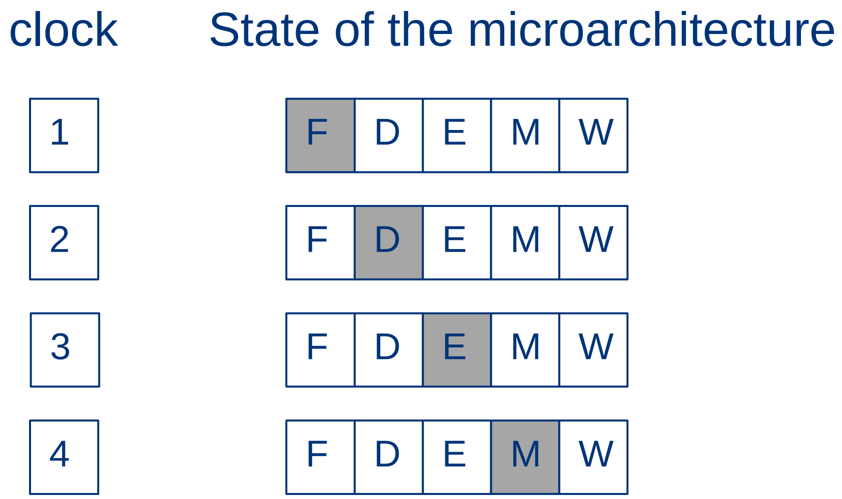
# Representations



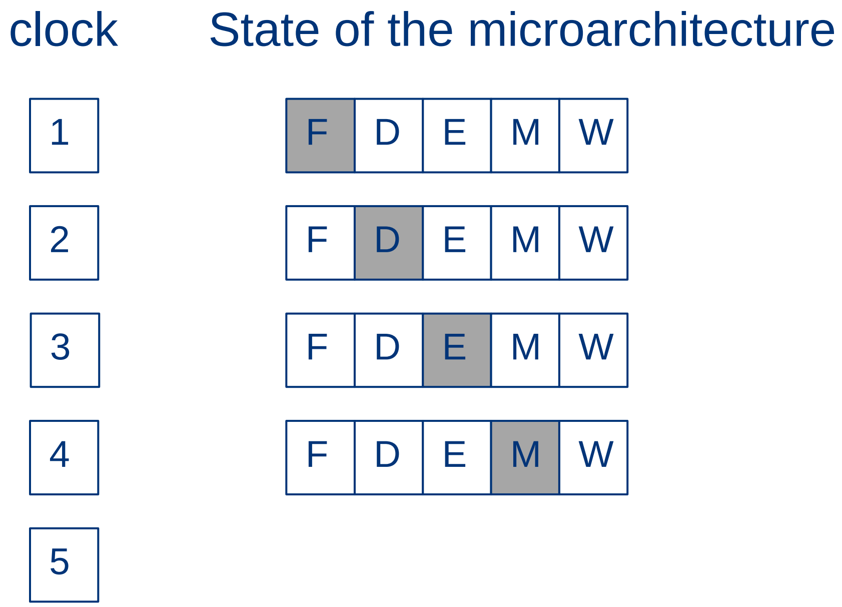
# Representations



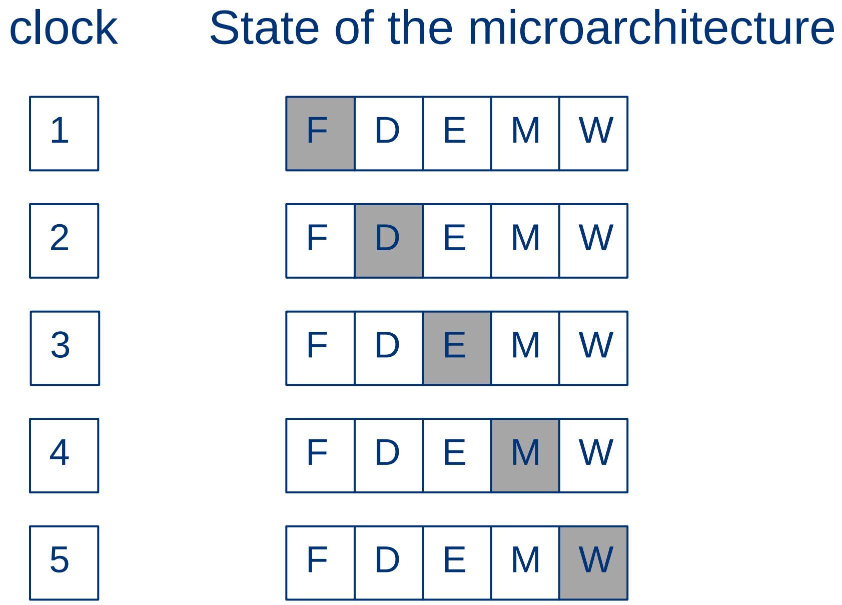
# Representations



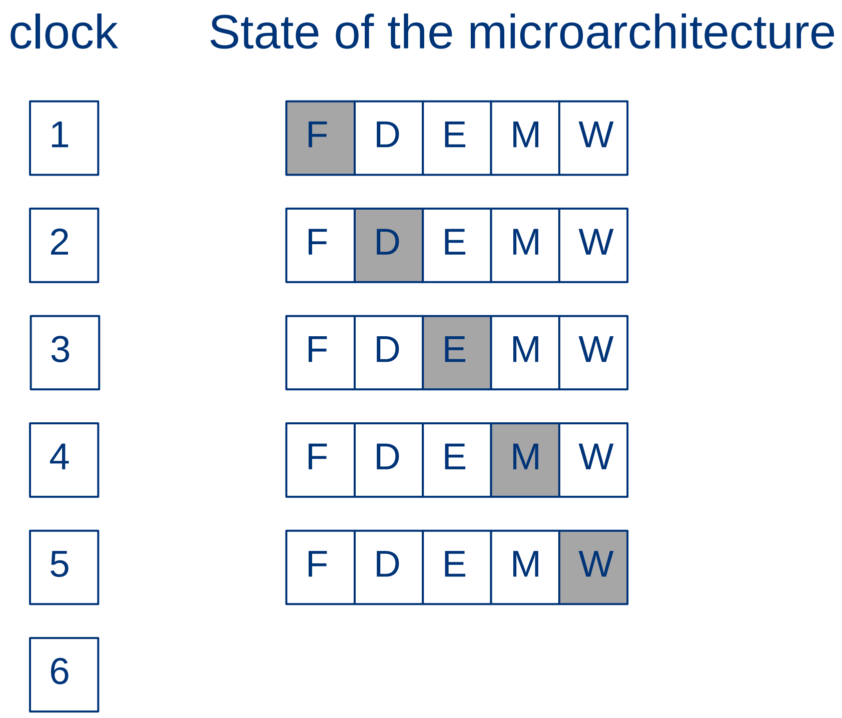
# Representations



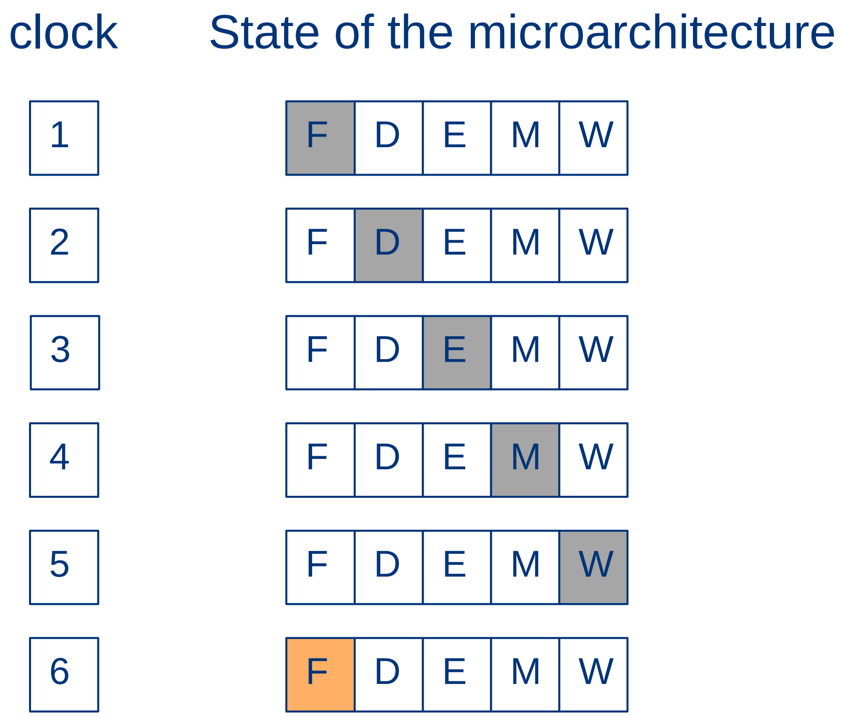
# Representations



# Representations

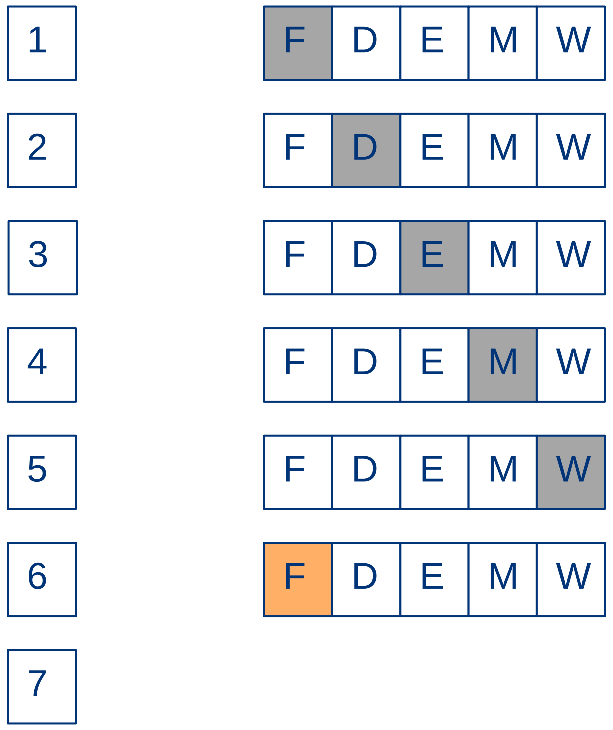


# Representations

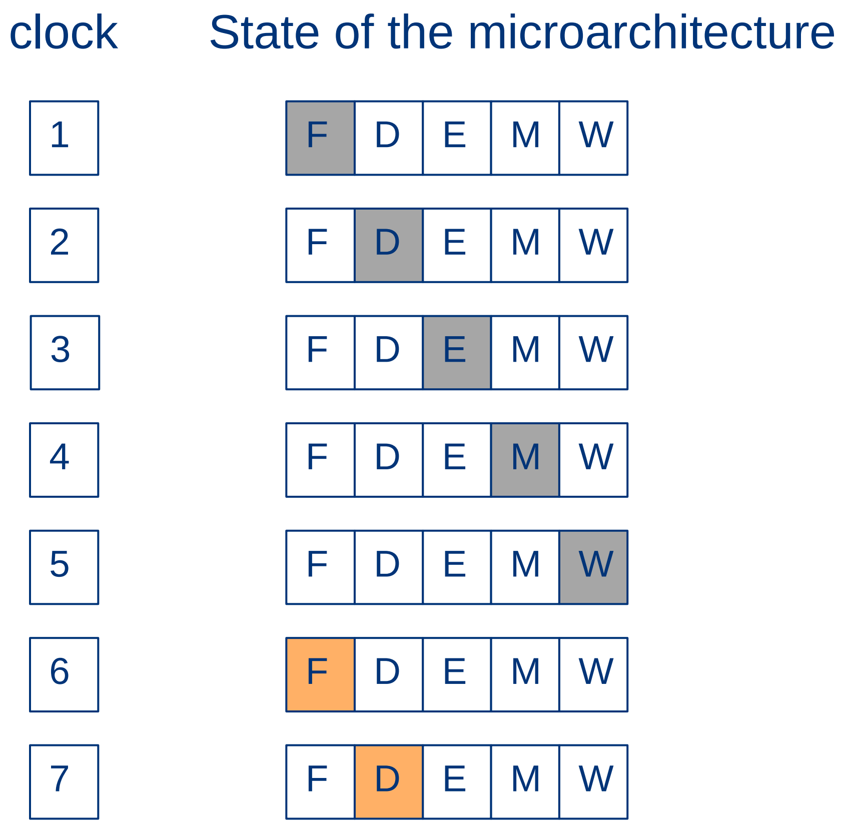


# Representations

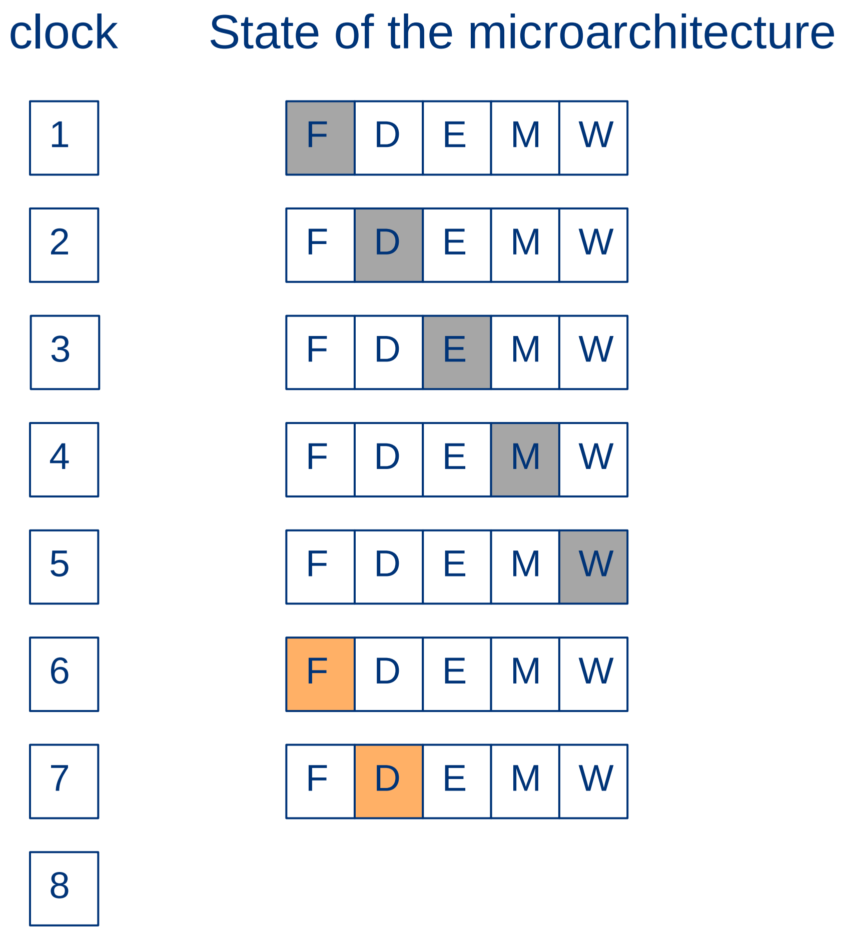
clock State of the microarchitecture



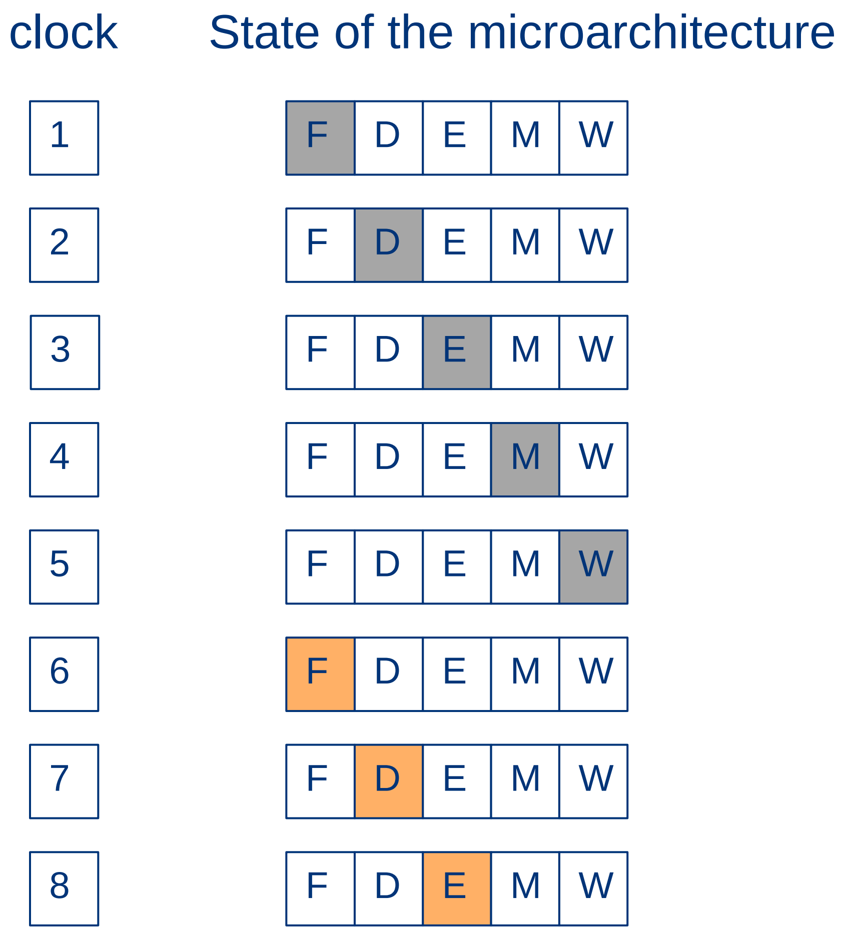
# Representations



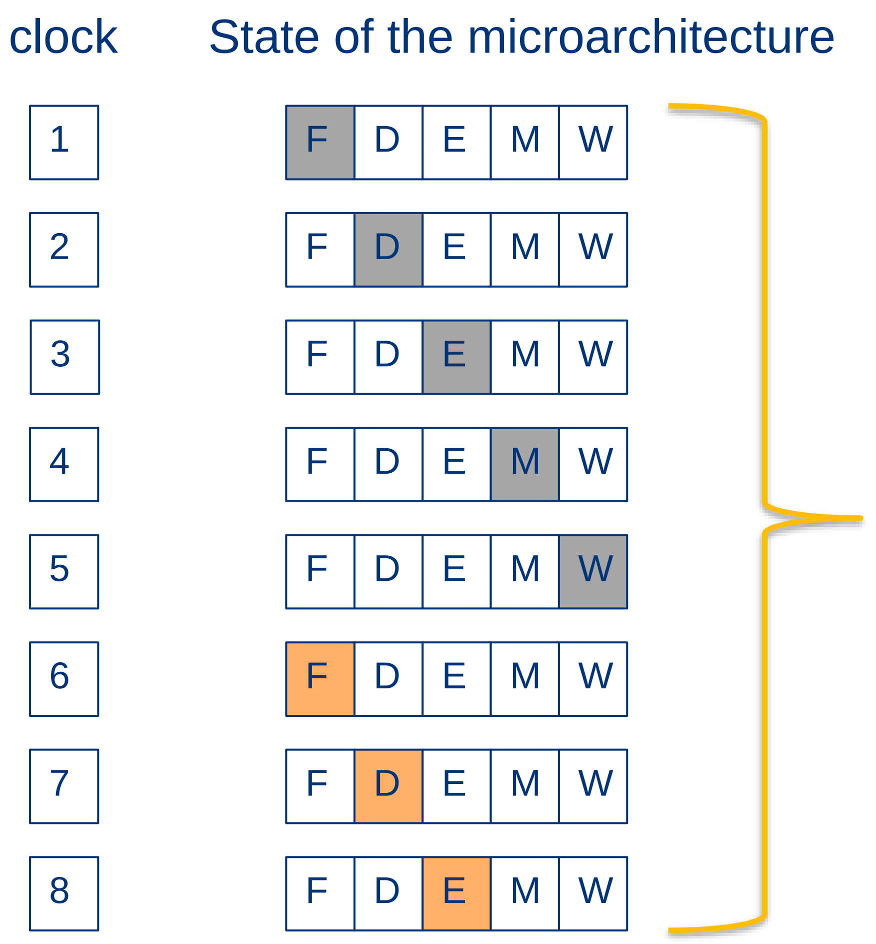
# Representations



# Representations



# Representations



Snapshot of the microarchitecture states and their utilization

Problem: Resources are under utilized
Solution: Pipelining

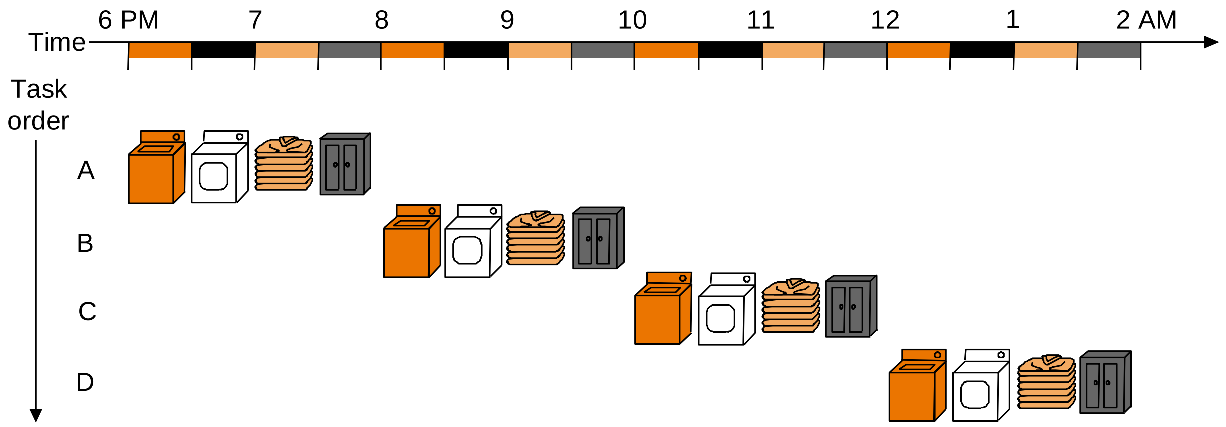
## Improve Performance of Multi-Cycle Microarchitecture

- **Can We Use the Idle Hardware to Improve Concurrency?**
  - Goal: More concurrency → Higher instruction throughput (i.e., more "work" completed in one cycle)
  - Idea: When an instruction is using some resources in its processing phase, process other instructions on idle resources not needed by that instruction
    - E.g., when an instruction is being decoded, fetch the next instruction
    - E.g., when an instruction is being executed, decode another instruction
    - E.g., when an instruction is accessing data memory (ld/st), execute the next instruction
    - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction
- **Pipelining ☺**

# Pipelining for instruction processing: Basic Idea

- **How to achieve concurrency in instruction processing?**
  - Divide the instruction processing cycle into distinct "stages" of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a different instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages
  - Process each stage in a single clock cycle
    - Pitfalls (later)
- **Benefit: Increases instruction processing throughput (1/CPI)**
- **Drawback: ?**
- **Quick question: Is pipelining a single-cycle or a multi-cycle microarchitecture?**

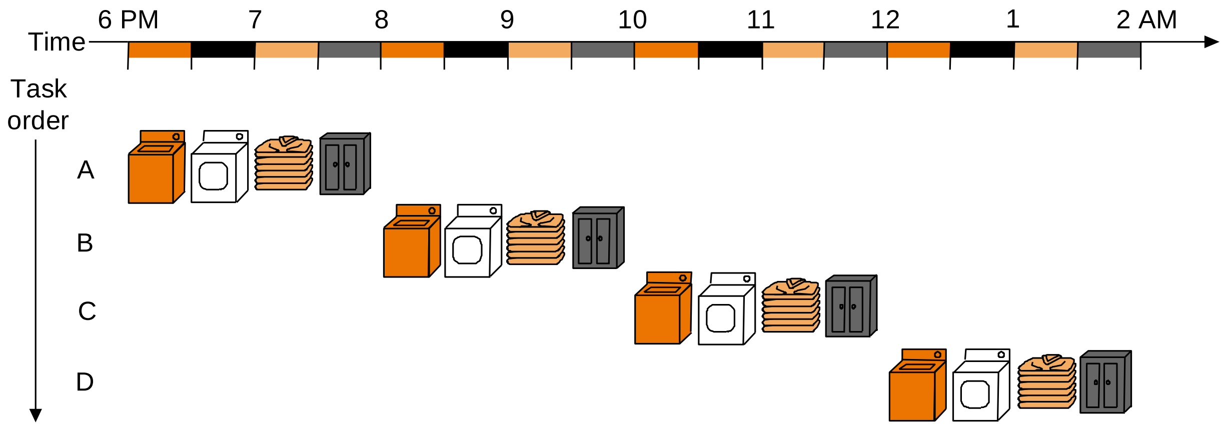
# The Laundry Analogy

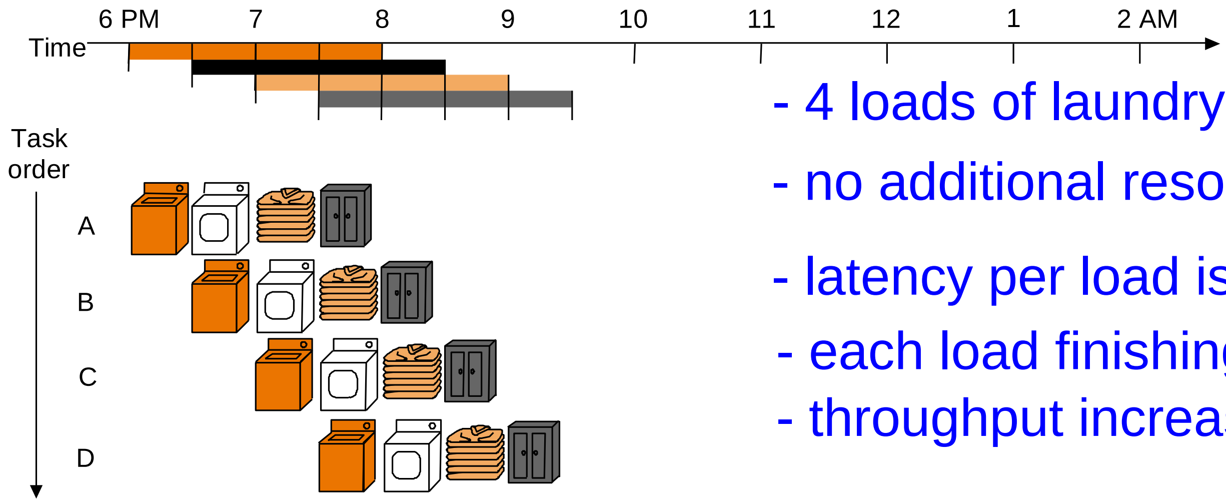


- **“place one dirty load of clothes in the washer”**
- **“when the washer is finished, place the wet load in the dryer”**
- **“when the dryer is finished, take out the dry load and fold”**
- **“when folding is finished, put the clothes in closet”**

- steps to do a load are sequentially dependent
- no dependence between different loads
- different steps do not share resources
- one load every 2 hours

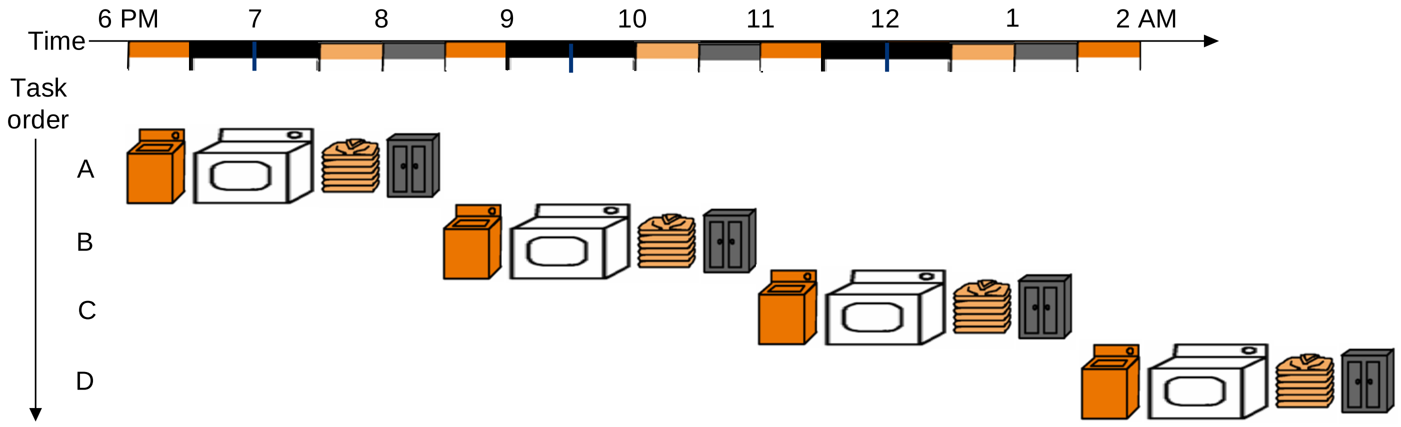
# The Laundry Analogy

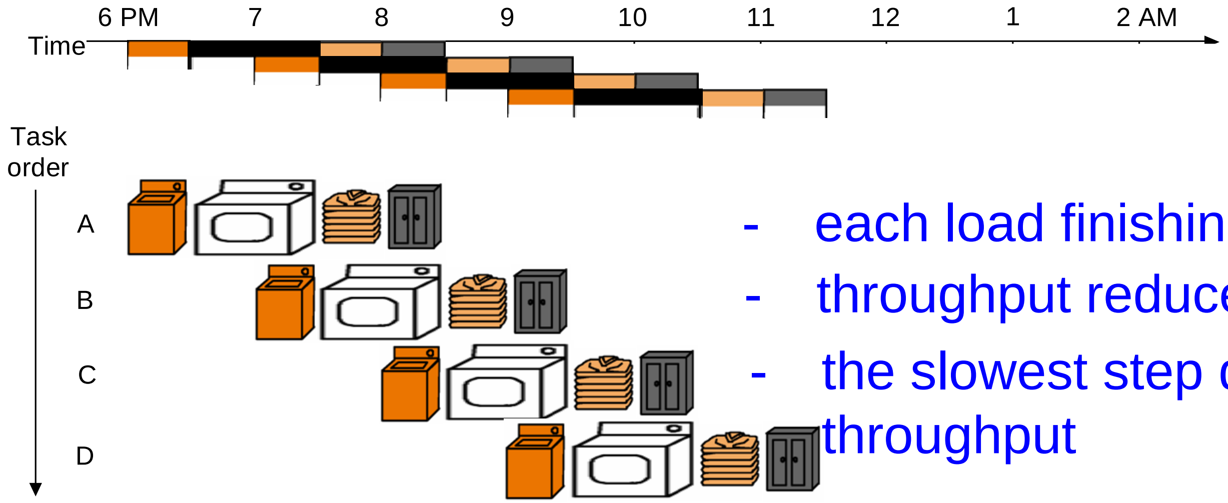




- 4 loads of laundry in parallel
- no additional resources
- latency per load is the same
- each load finishing every half hou
- throughput increased by 4

# Pipelining Multiple Loads of Laundry: In Practice





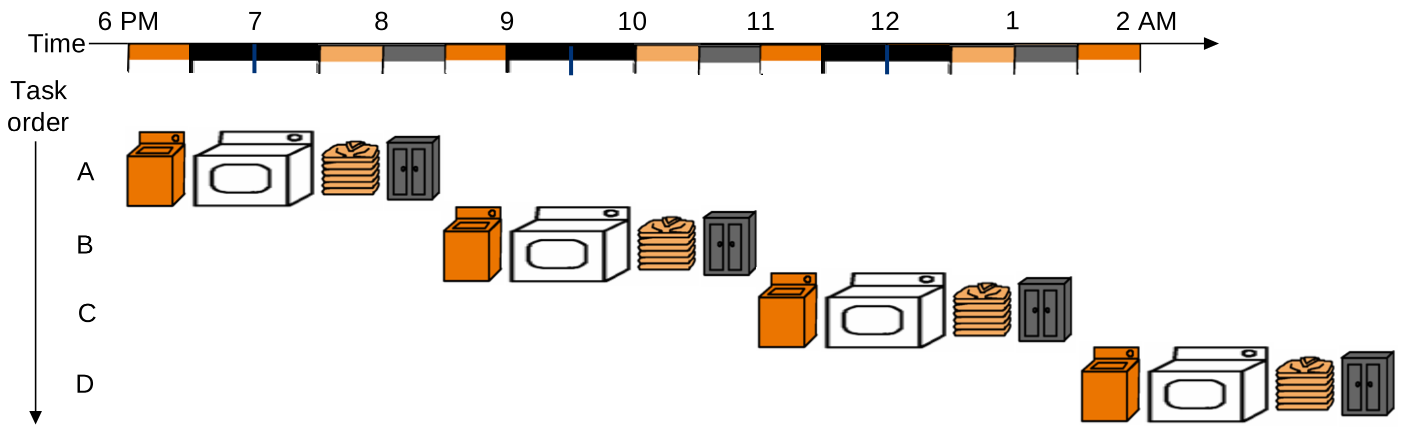
- each load finishing every hour
- throughput reduced by ½
- the slowest step decides throughput

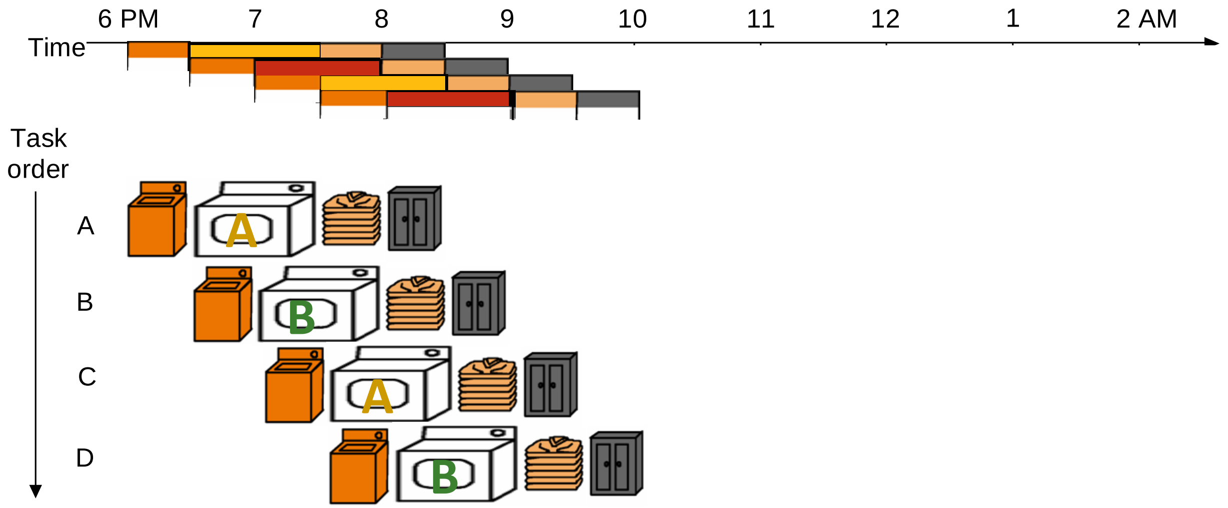
# An Ideal Pipeline

- **Increase throughput with little increase in cost**
  - hardware cost, in case of instruction processing

- **Repetition of identical operations**
  - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)
- **Repetition of independent operations**
  - No dependencies between repeated operations
- **Uniformly partitionable suboperations**
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
  - In the previous example, everything takes 30 mins

# Pipelining Multiple Loads of Laundry: In Practice

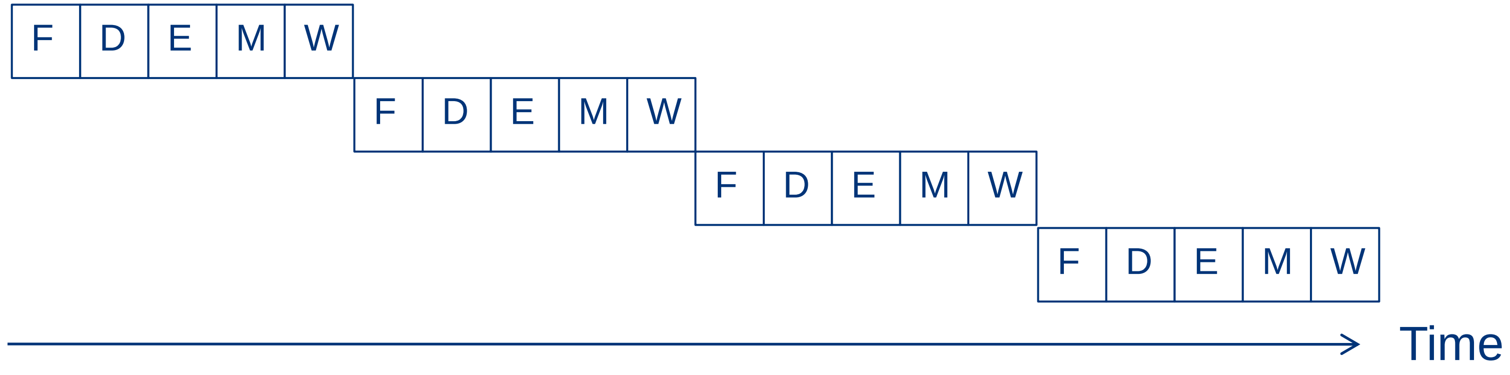




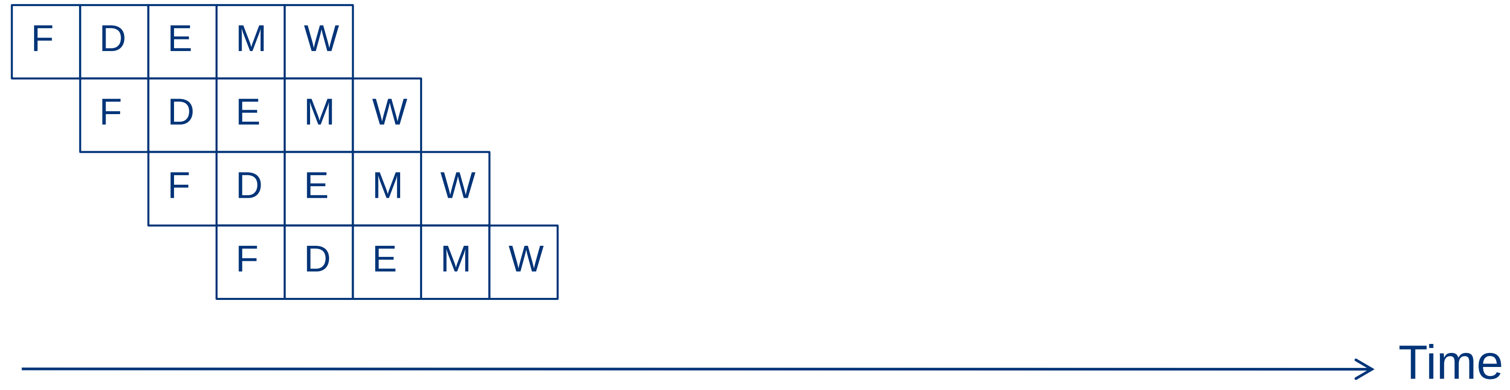
throughput restored (2 loads per hour) using 2 dryers

# Pipelining for Instruction Processing

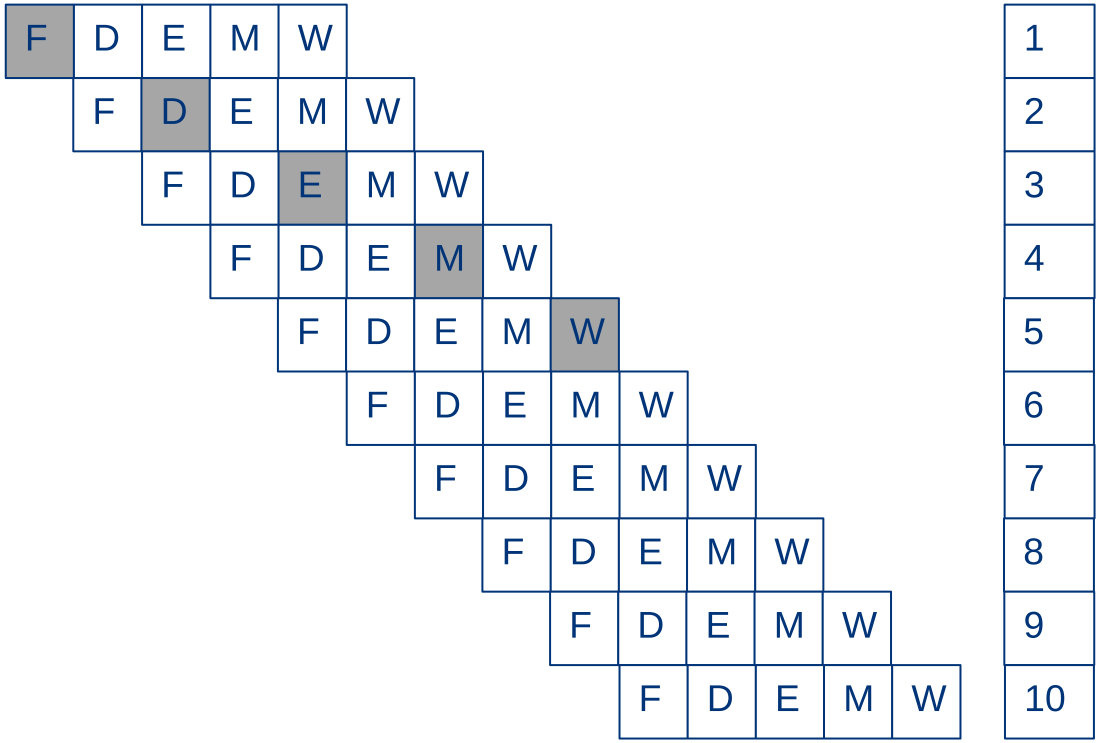
- **Multi-cycle: 5 cycles per instruction**



- **Pipelined:**

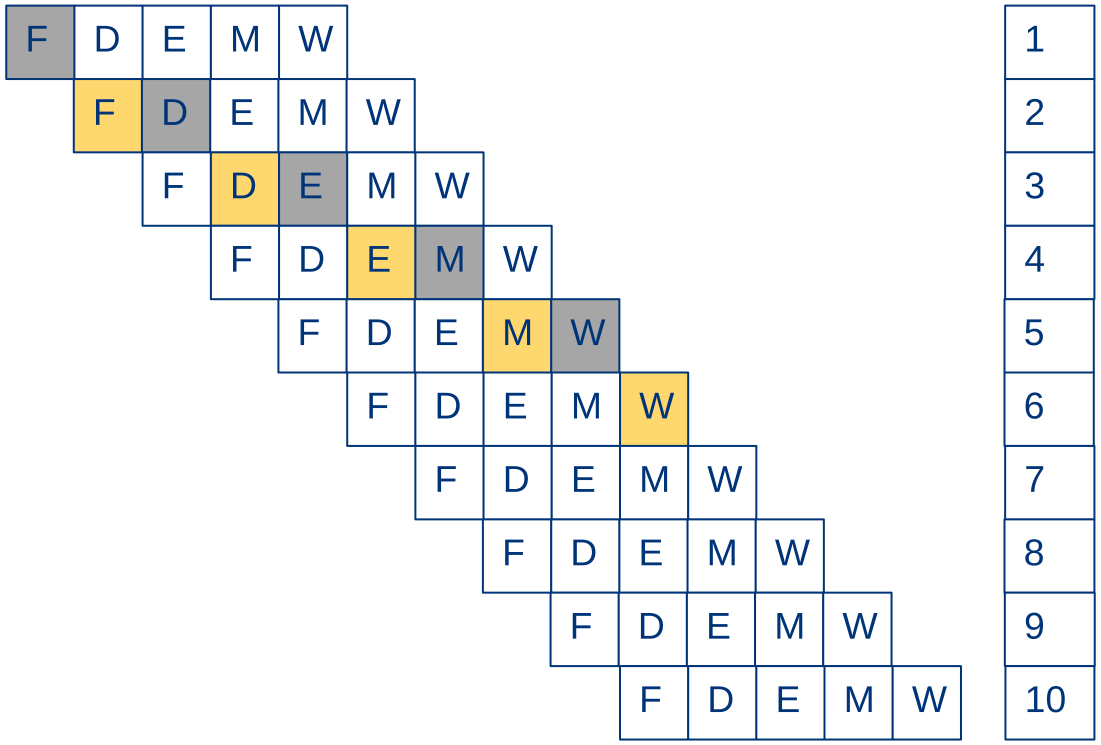


# Example: Execution of Independent Instructions

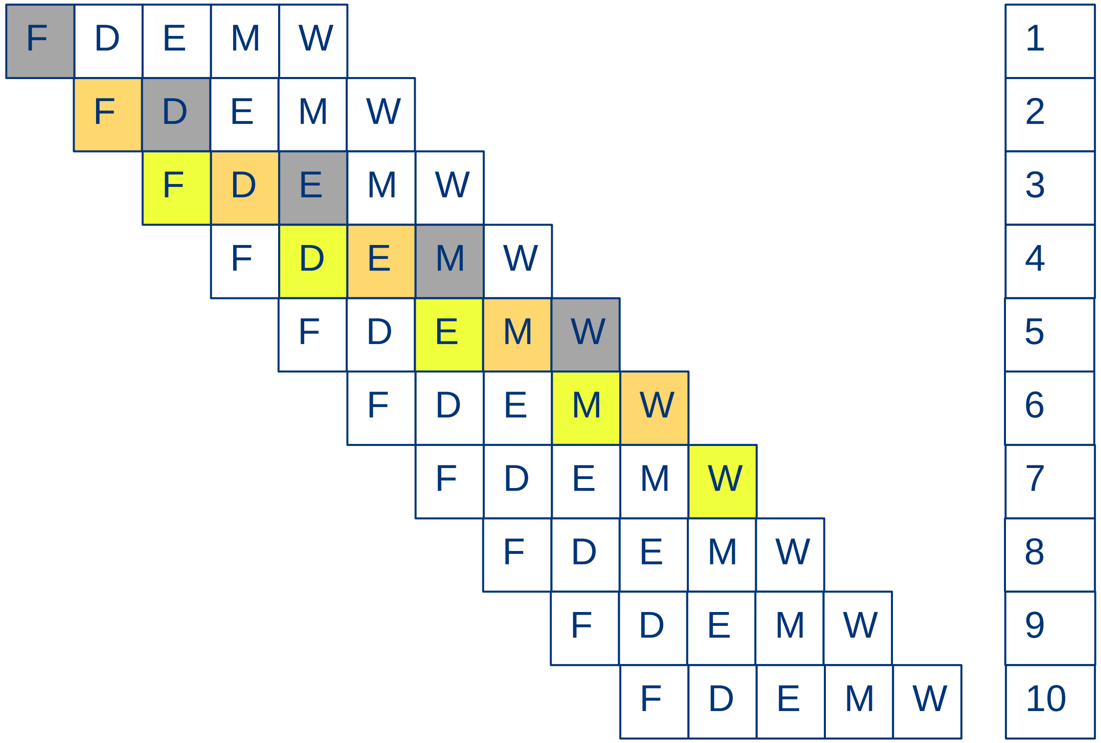


| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **F** | D | E | M | W | | | | | | | | | | 1 |
| | F | **D** | E | M | W | | | | | | | | | 2 |
| | | F | D | **E** | M | W | | | | | | | | 3 |
| | | | F | D | E | **M** | W | | | | | | | 4 |
| | | | | F | D | E | M | **W** | | | | | | 5 |
| | | | | | F | D | E | M | W | | | | | 6 |
| | | | | | | F | D | E | M | W | | | | 7 |
| | | | | | | | F | D | E | M | W | | | 8 |
| | | | | | | | | F | D | E | M | W | | 9 |
| | | | | | | | | | F | D | E | M | W | 10 |

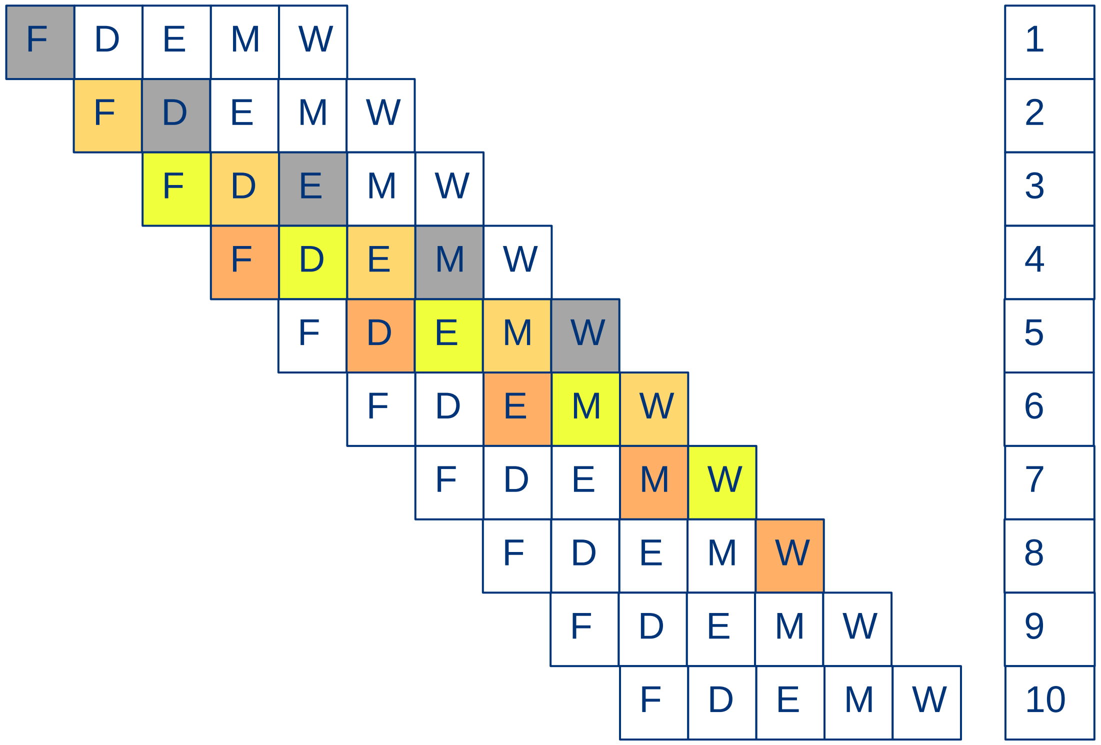
# Example: Execution of Independent Instructions



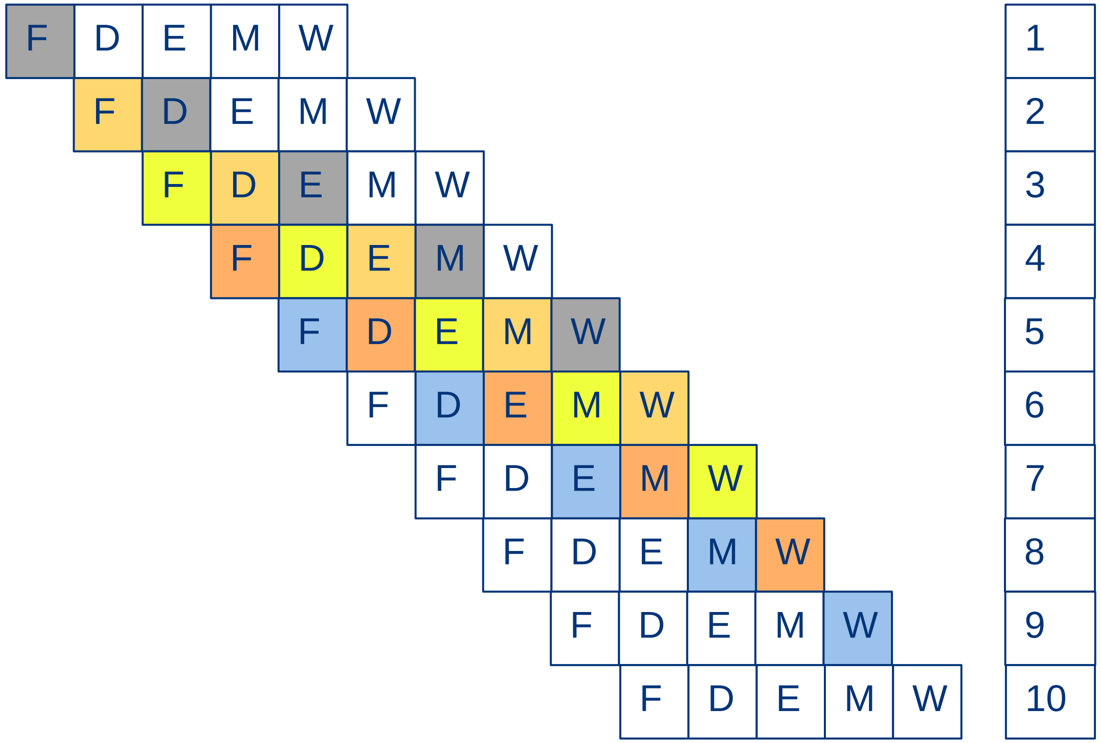
# Example: Execution of Independent Instructions



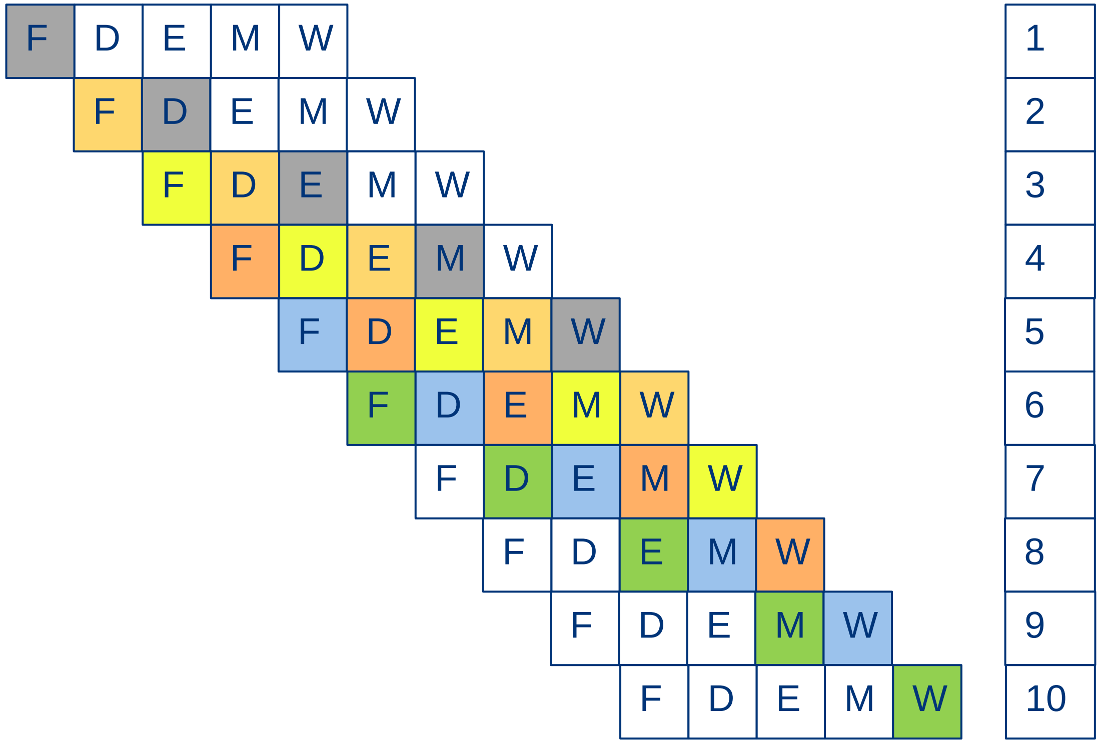
# Example: Execution of Independent Instructions



# Example: Execution of Independent Instructions

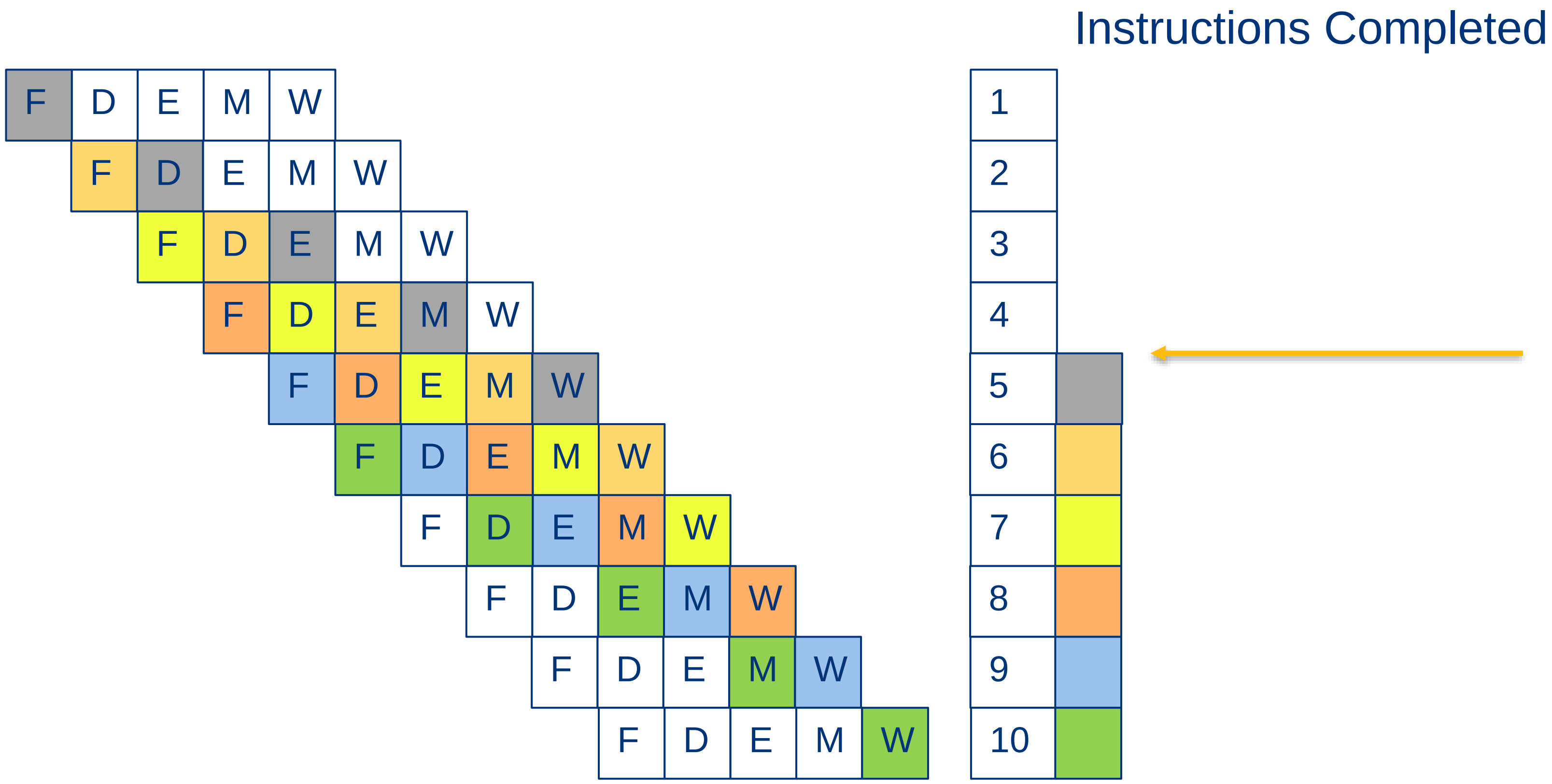


# Example: Execution of Independent Instructions



## Pipelining: CPI

# Example: Execution of Independent Instructions



# Key Facts about Pipelining

- **There are no instructions completed for the first 4 cycles**
- **Once all the stages are working (i.e., the pipeline is full), instructions starts getting completed**
  - We call the state where the pipeline becomes full as the steady state
  - IPC/CPI are computed in steady state
- **What are the steady state metrices?**
  - CPI?
    - 1
  - IPC?
    - 1
  - What about instruction latency?
    - 5 cycles
  - Does the instruction latency greater than, equal to, or lower than that of multi-cycle microarchitecture?

## Revisit

- **Multi-cycled vs pipelined microarchitecture**
  - Both has multiple “stages” between the initial and final “states” which are clocked
  - Multi-cycled microarchitecture
    - CPI > 1, IPC < 1
  - Pipelined microarchitecture
    - CPI = 1 and IPC = 1

- **Lets keep them as separate categories**

# Pipelined Microarchitecture: Performance

- **CPI = 1**
- **Clock Cycle Time = 200ps**
- **Execution time of 100 billion instructions**
  - = {# of instructions} x {Average CPI} x {clock cycle time}
  - = {100 x $10^{9}$} x 1 x {200 x $10^{-12}$}
  - = 20 seconds
- **Average latency of each instruction = 5 cycles = 5 x 200 = 1000ps**
- **Average instruction throughput = 1 / (1 x 200 x $10^{-12}$) = 5 billion instructions per second**
- **IPC = 1 / CPI = 1**

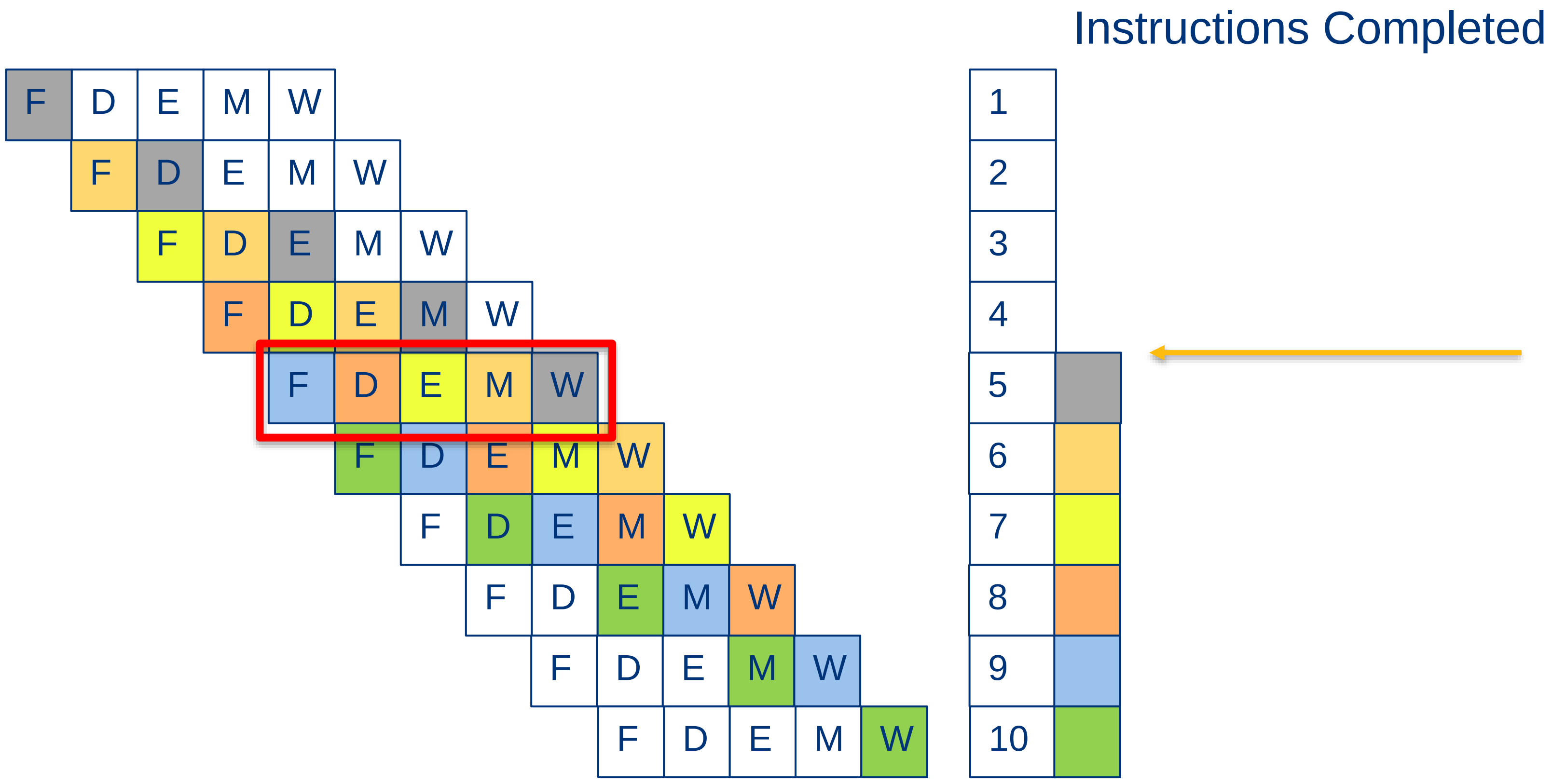
## Recap of Multi-Cycle Microarchitecture

| | Single cycle | Multi-cycle | |
|---|---|---|---|
| CPI | 1 | 4.12 | |
| Clock cycle time | 900ps | 200ps | |
| Execution time of 100 billion instructions | 90s | 82.4s | |
| Average latency of each instruction | 900ps | 824ps | |
| Average instruction throughput | 1.1 bips | 1.21 bips | |
| IPC | = 1/1 = 1 | = 1/4.12 = 0.24 | |

# Recap of Multi-Cycle Microarchitecture

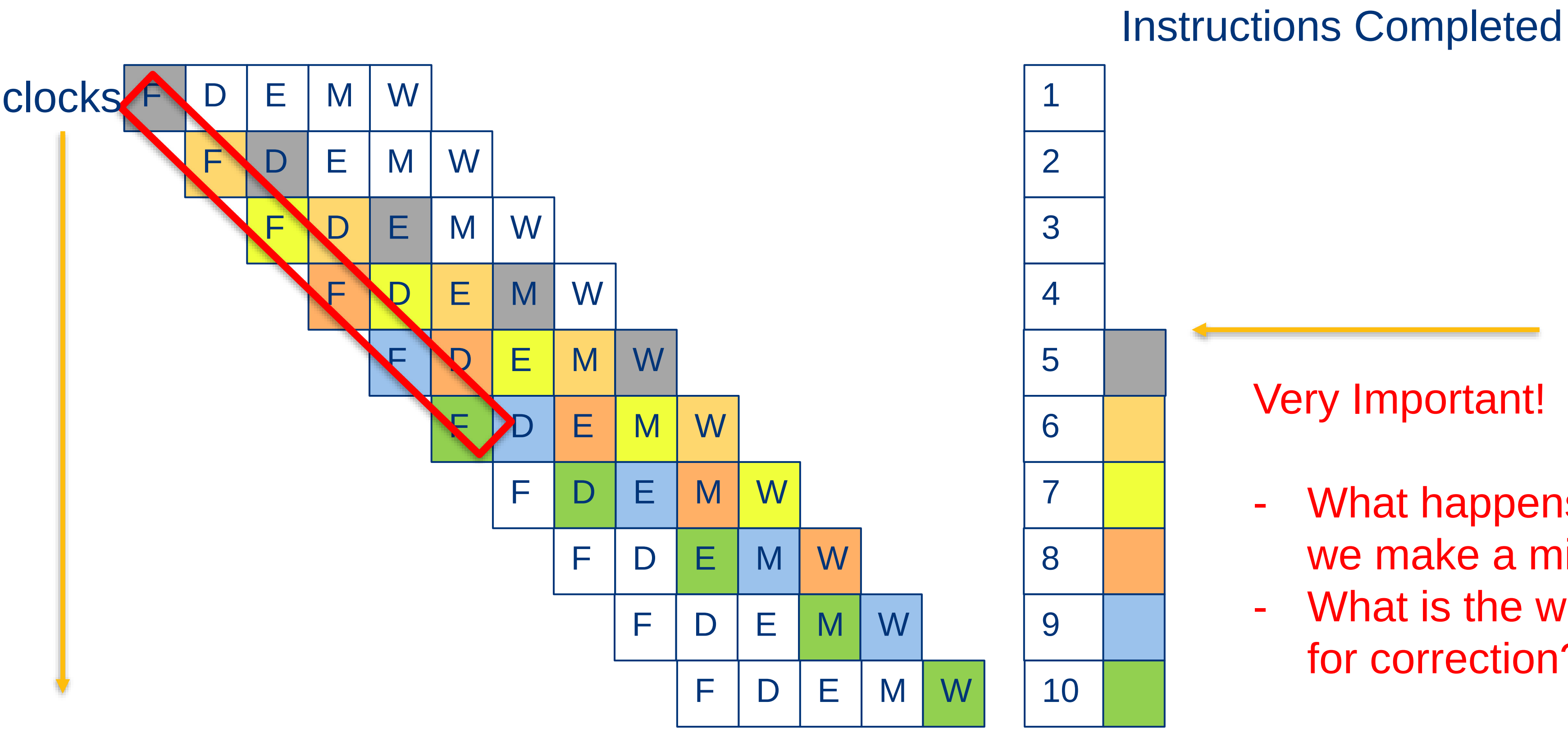
| | Single cycle | Multi-cycle | Pipeline |
|---|---|---|---|
| CPI | 1 | 4.12 | 1 |
| Clock cycle time | 900ps | 200ps | 200ps |
| Execution time of 100 billion instructions | 90s | 82.4s | 20s |
| Average latency of each instruction | 900ps | 824ps | 1000ps |
| Average instruction throughput | 1.1 bips | 1.21 bips | 5 bips |
| IPC | 1/1 = 1 | 1/4.12 = 0.24 | 1/1 = 1 |

# Example: Execution of Independent Instructions



Only a part of the instruction gets processed at any given clock cycle
Different stages process different instructions (part of it) at any given clock cycle

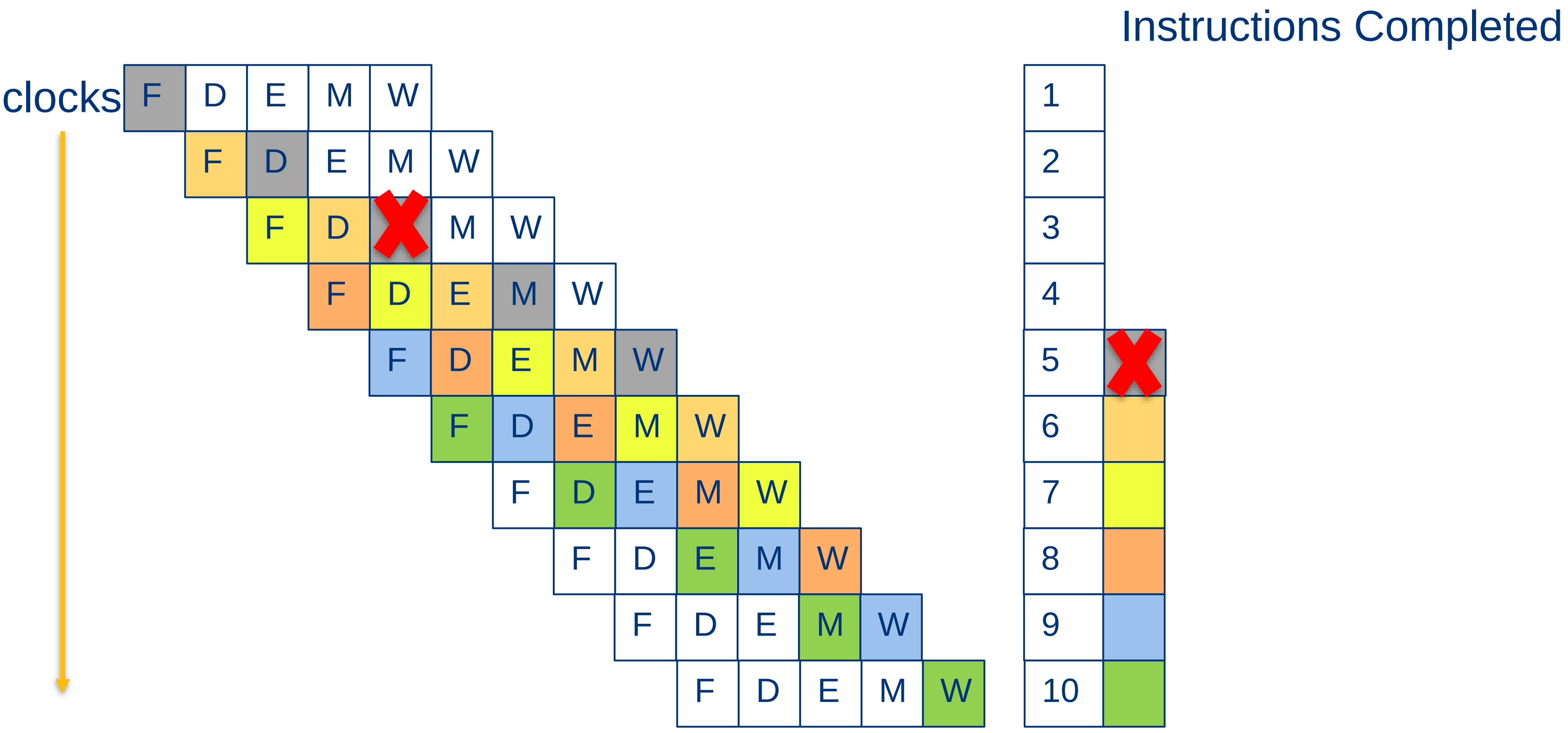
# Example: Execution of Independent Instructions



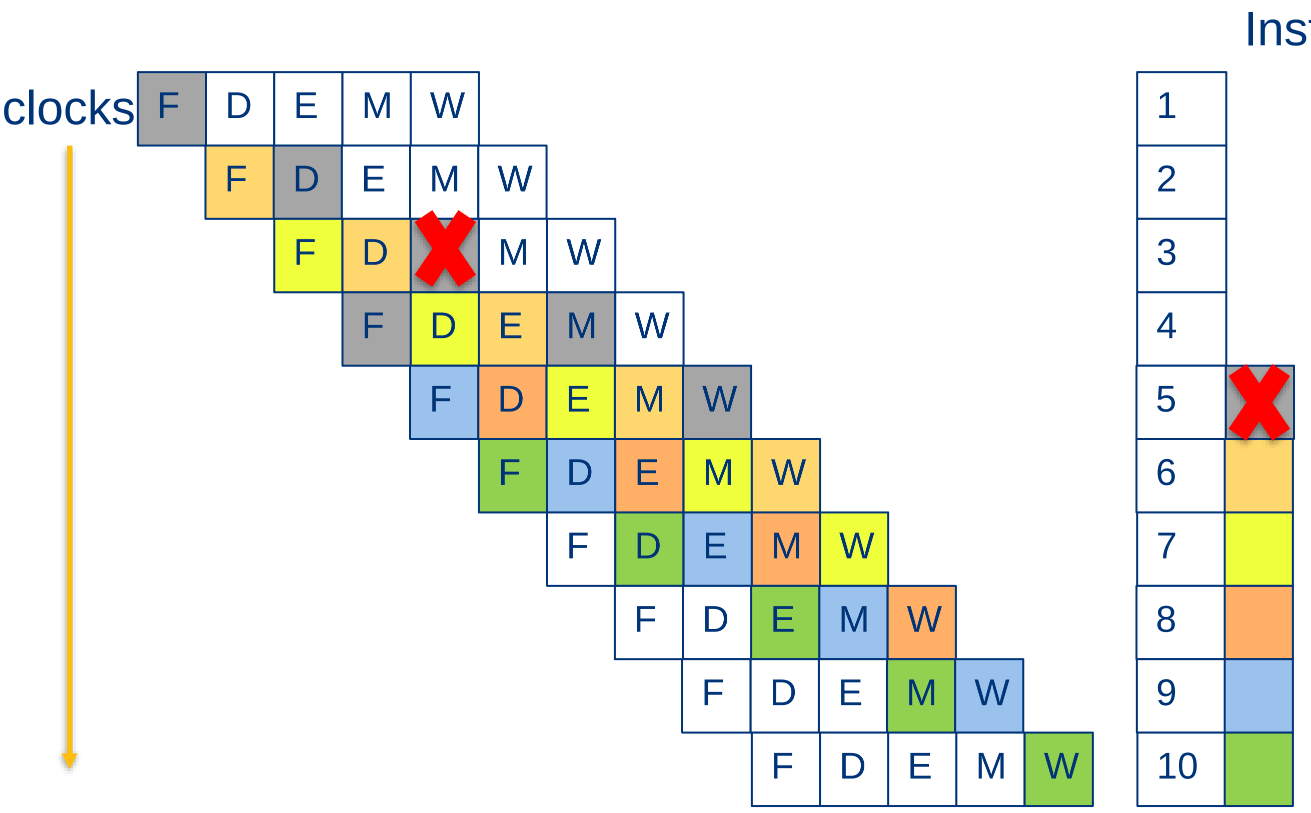
Very Important!

- What happens when we make a mistake?
- What is the window for correction?

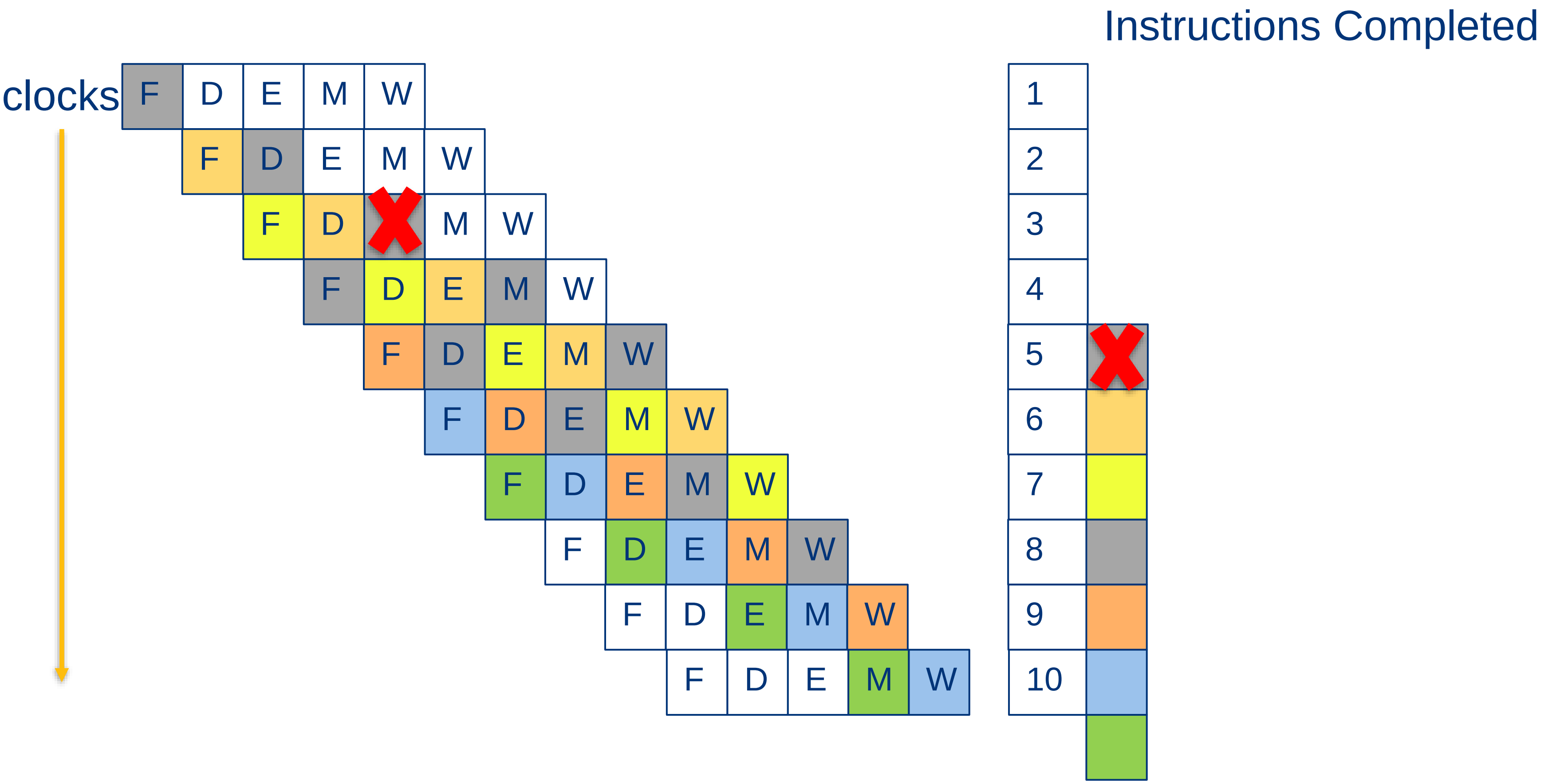
# Example: Execution of Instructions



# Example: Execution of Instructions

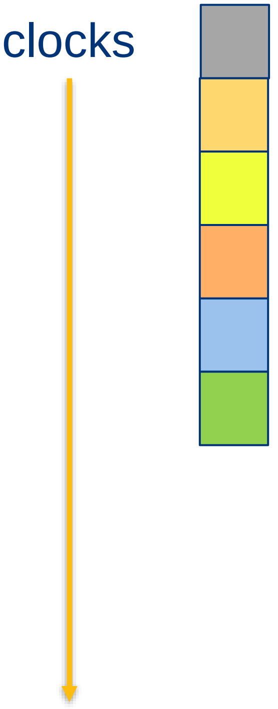


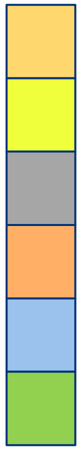
# Example: Execution of Instructions



# What does the programmer see?

- **Expected Instruction Order**
- **Obtained Instruction Order**

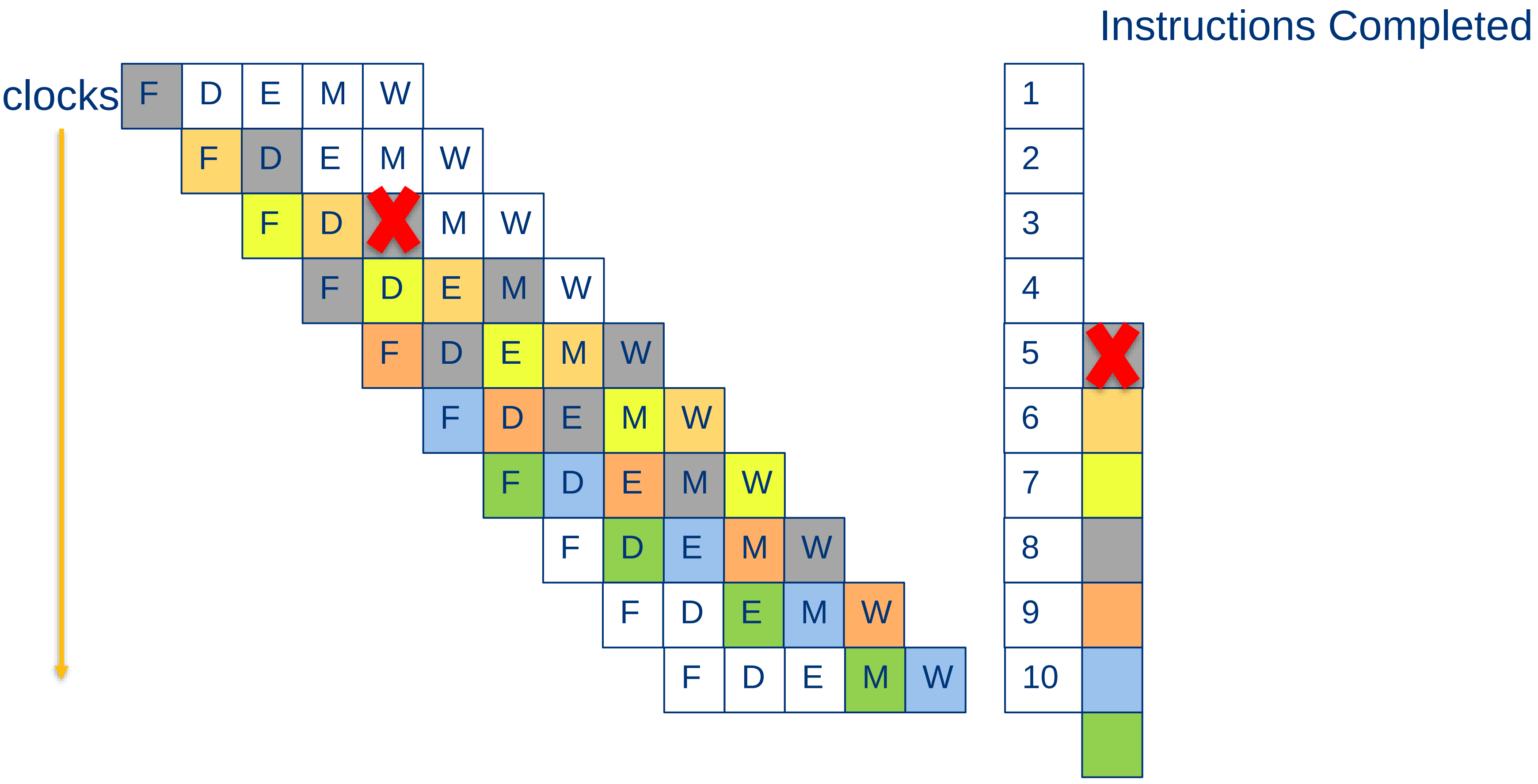




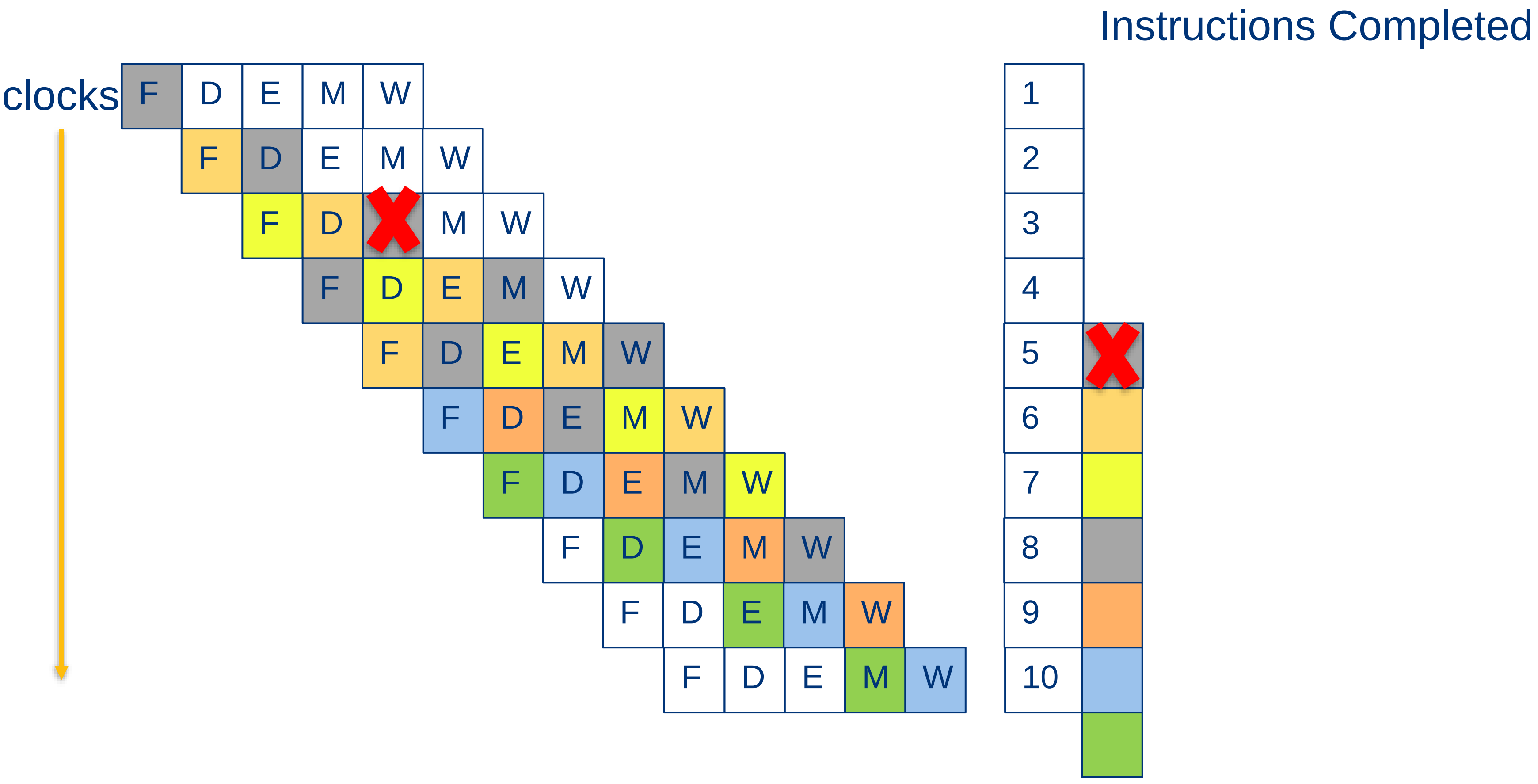
Do you see a difference in instruction order?

Not acceptable!

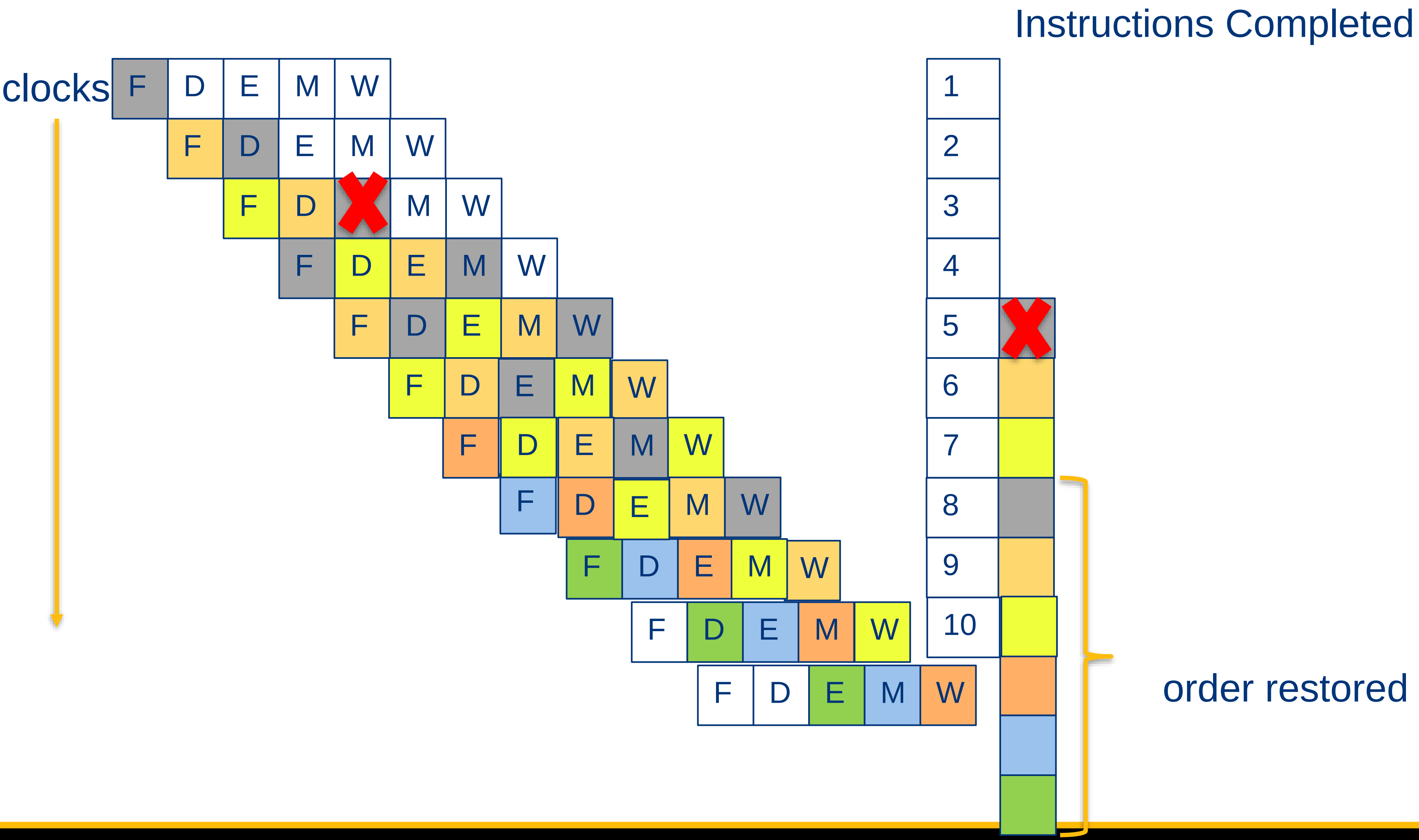
# Example: Execution of Instructions



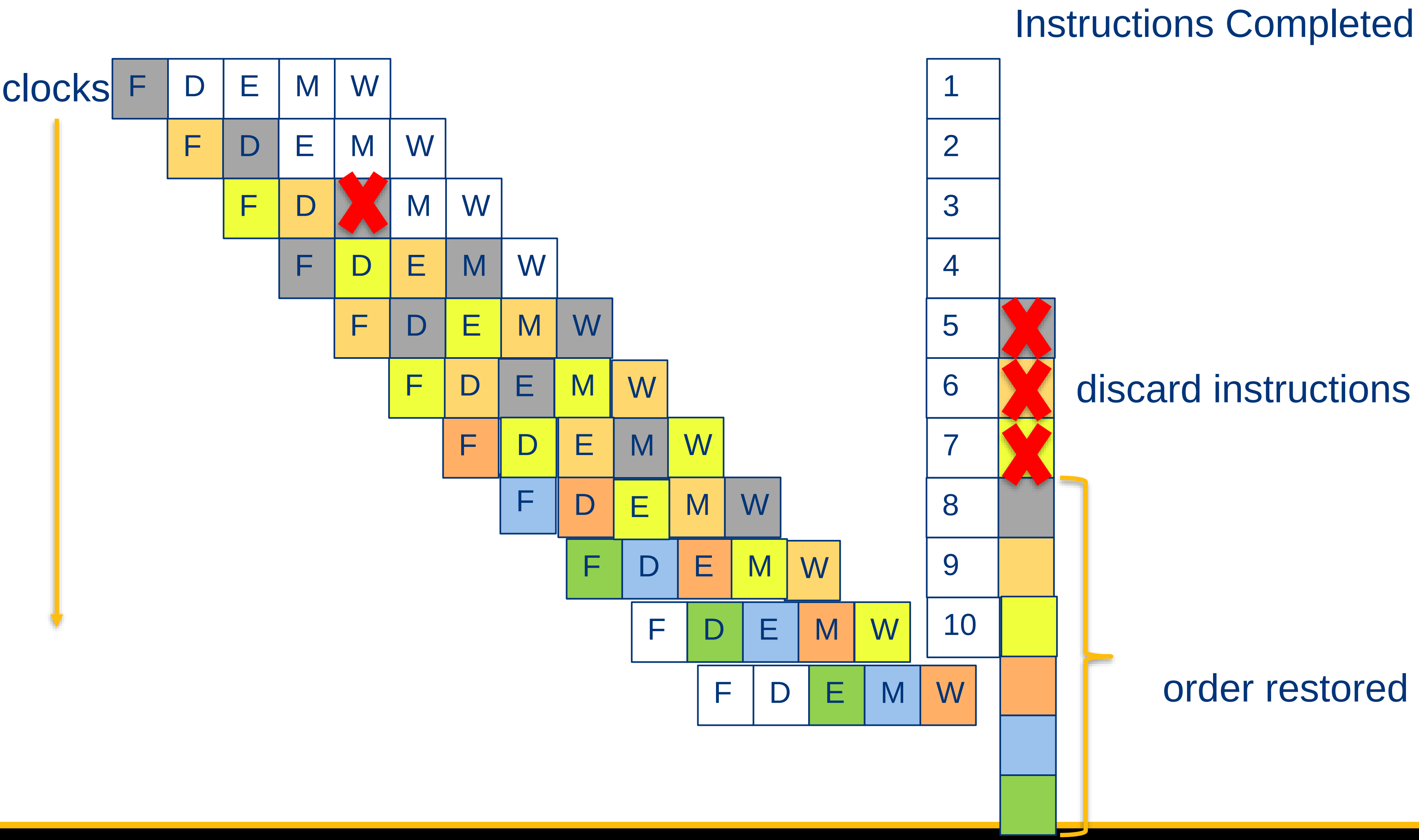
# Example: Execution of Instructions



# Example: Execution of Instructions



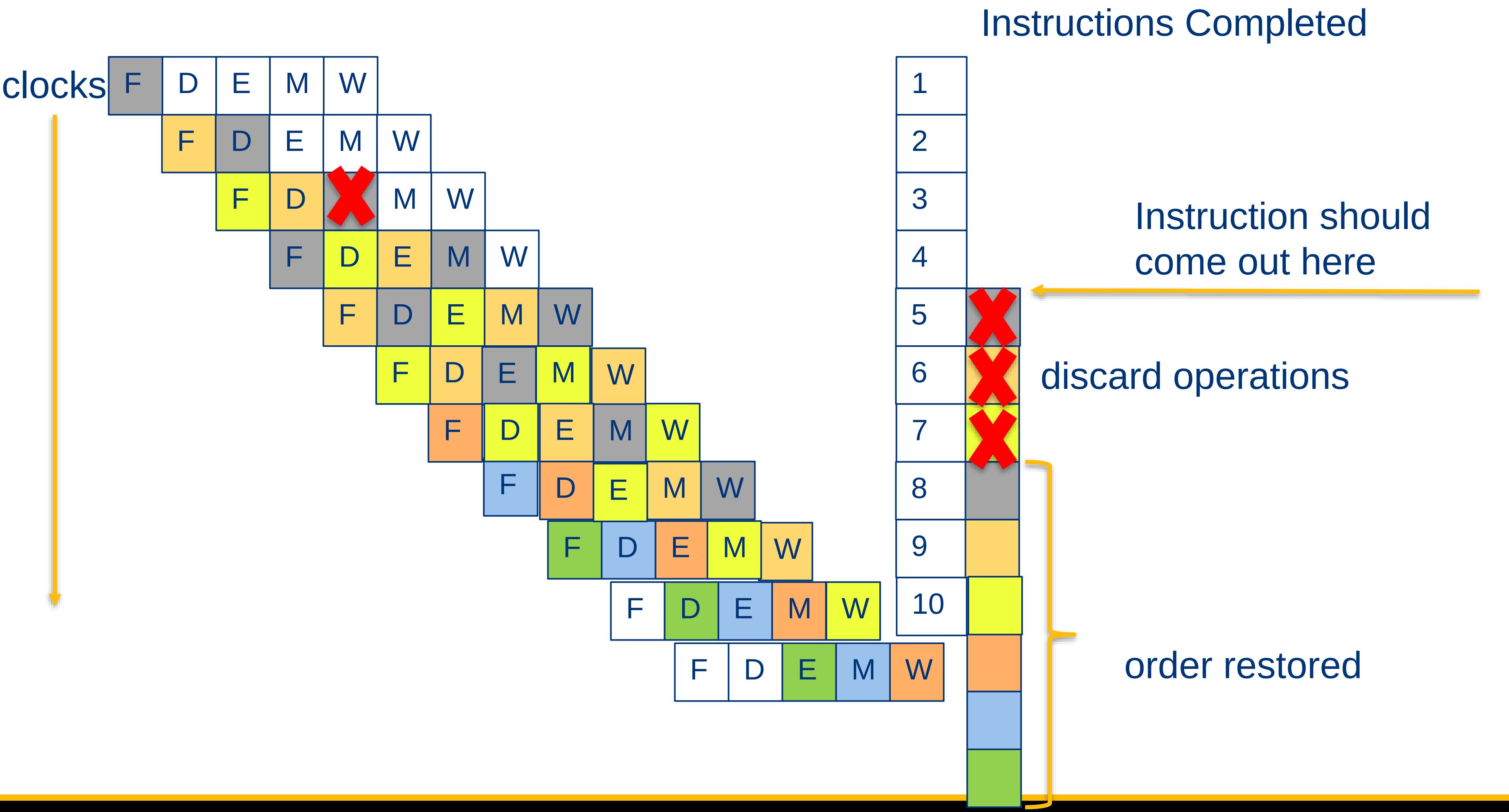
# Example: Execution of Instructions



## Incorrect Execution

- **Discarding all instructions that are currently in the pipeline**
  - Also called pipeline flushing
  - All stages of the pipeline are processing some instruction
  - Important: Only those instructions are flushed which are in and prior to the stage where error happened!!
  - How many total instructions discarded?
    - 3
  - On what this number "3" depends on?
    - The stage where error happened
  - If the error happened in the decode stage, how many instructions will be flushed?
    - 2

# Example: Execution of Instructions



# Incorrect Execution

- **Discarding all instructions that are currently in the pipeline**
  - Look at the time to process instruction. How many cycles are there where no instruction is completed?
    - 3
  - On what does it depends?
    - The stage where error happened
  - All cycles where there is no valid instruction output, we call stall cycles or bubbles
  - How many stall cycles?
    - 3

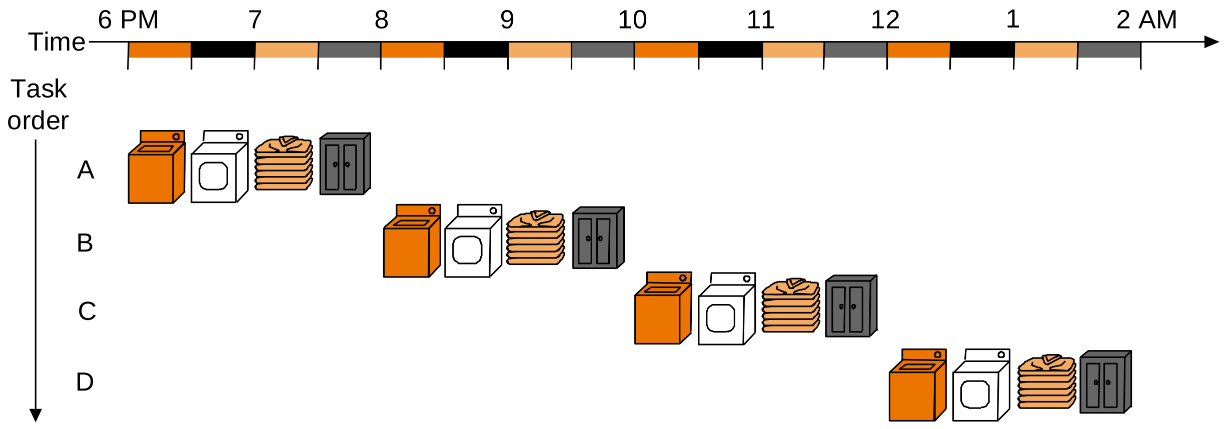
## Impact of Bubbles in pipeline

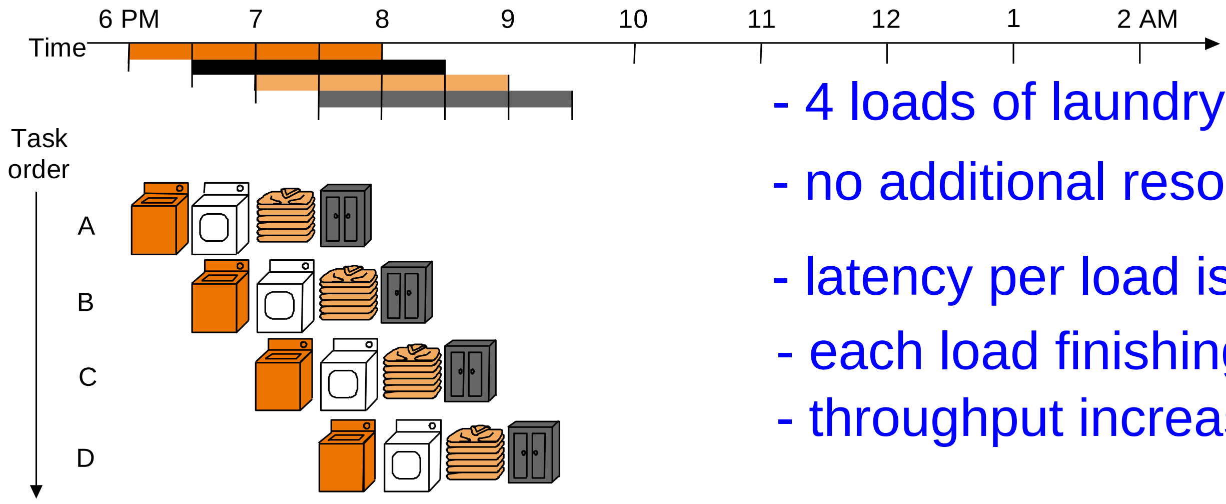
- **100 instructions**
  - No bubbles
    - 100 instructions in 100 cycles (steady state)
      - CPI = 100/100 = 1
  - 1 instruction in error (3 bubbles)
    - 100 instructions in 103 cycles (steady state)
      - CPI = 103/100 = 1.03
  - 10 instructions in error (3 bubbles/incorrect instruction)
    - 100 instructions in 130 cycles (steady state)
      - CPI = 130/100 = 1.3
- **Observe how CPI is climbing up**
  - High CPI is bad

# Pipeline bubbles concluding remarks

- Pipeline bubbles are bad and must be avoided
- Bubbles once introduced, stays in the pipeline

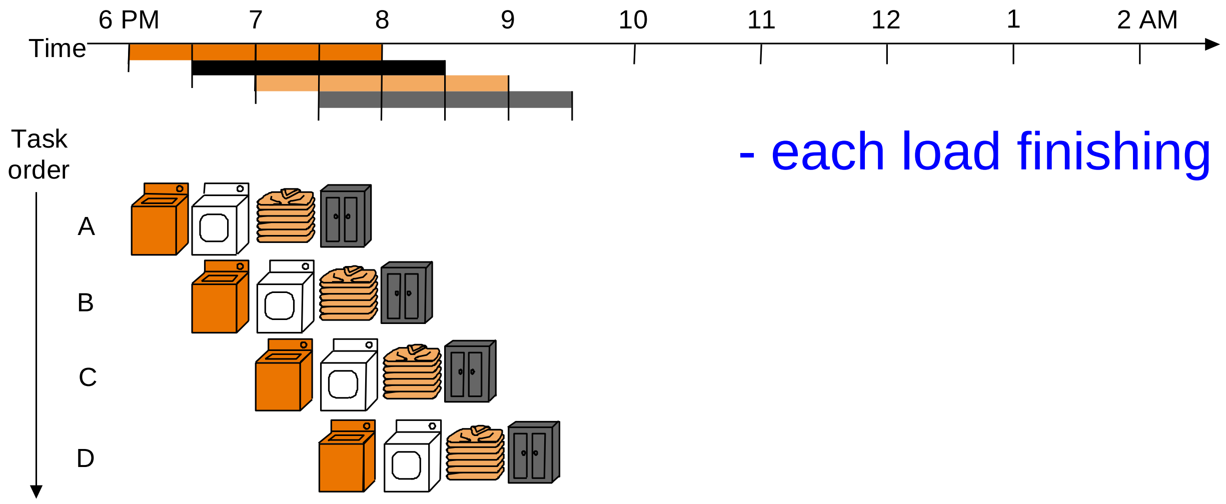
# The Laundry Analogy



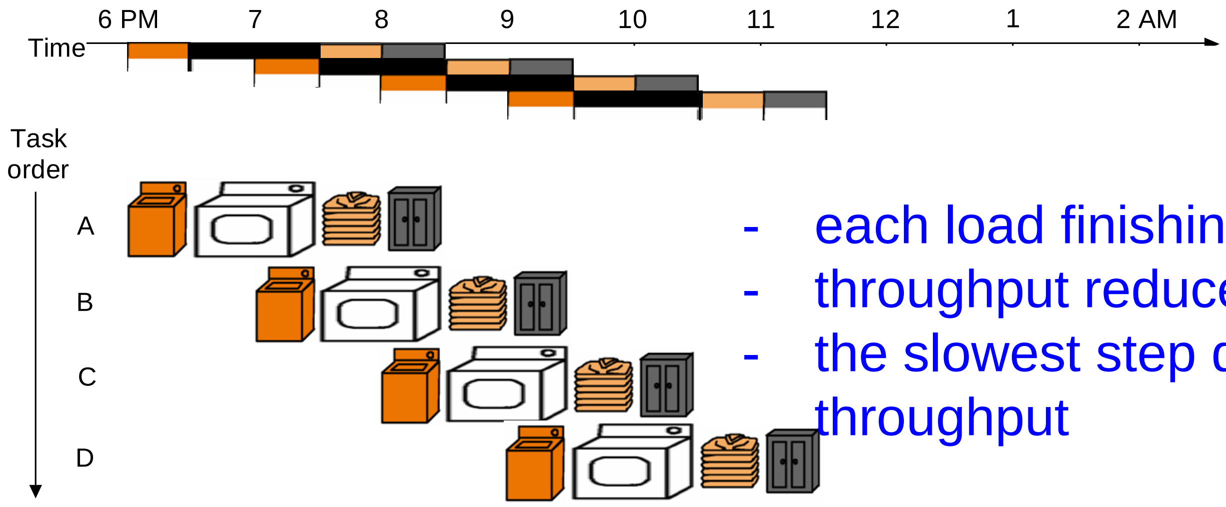


- 4 loads of laundry in parallel
- no additional resources
- latency per load is the same
- each load finishing every half hou
- throughput increased by 4

# Pipelining Multiple Loads of Laundry: In Practice



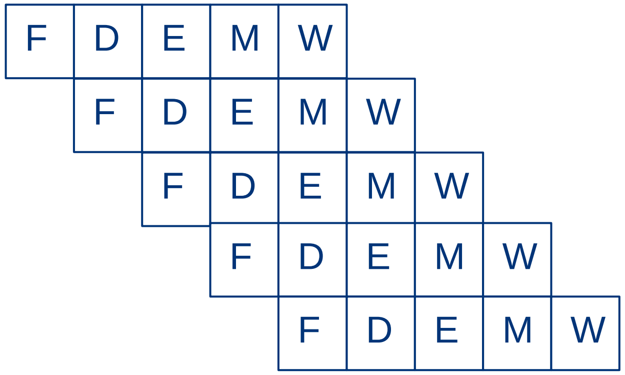
- each load finishing every half hour



- each load finishing every hour
- throughput reduced by ½
- the slowest step decides throughput

# Theoretical Analysis

- **5-stage pipeline**



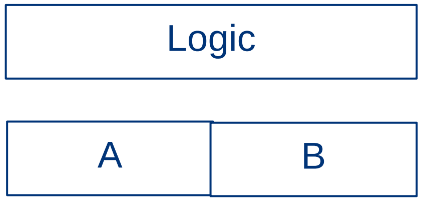
Throughput = 1 / delay of stage

## Theoretical Analysis

- **2-stage pipeline**

## Theoretical Analysis

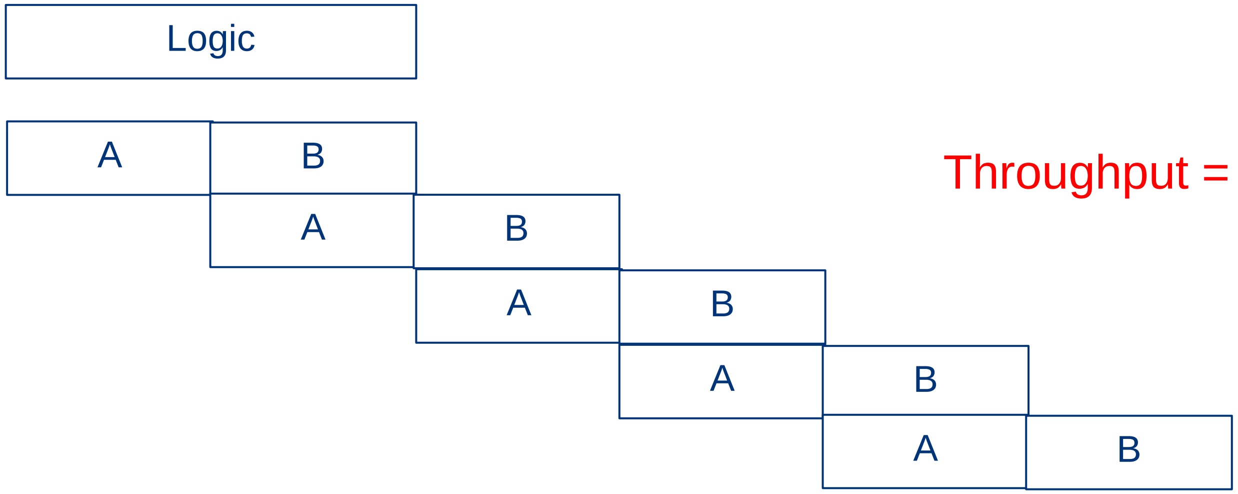
- **2-stage pipeline**



Throughput = 1 / delay of stage

# Theoretical Analysis

- **2-stage pipeline**



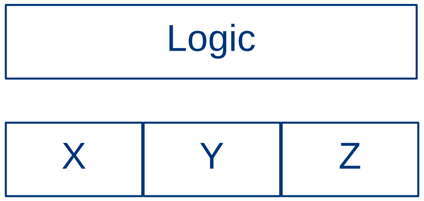
Throughput = 1 / delay of stage

## Theoretical Analysis

- **3-stage pipeline**

## Theoretical Analysis

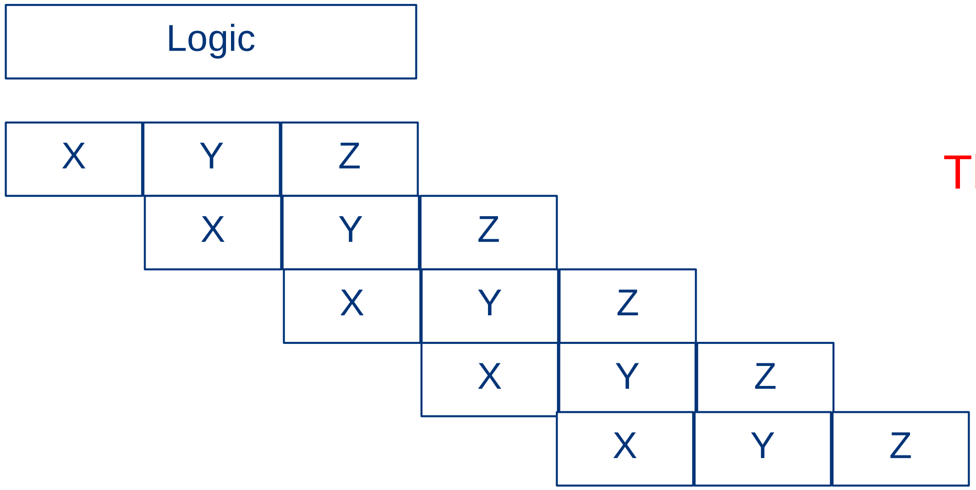
- **3-stage pipeline**



Throughput = 1 / delay of stage

## Theoretical Analysis

- **3-stage pipeline**



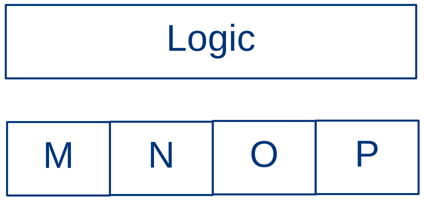
Throughput = 1 / delay of stage

# Theoretical Analysis

- **4-stage pipeline**

## Theoretical Analysis

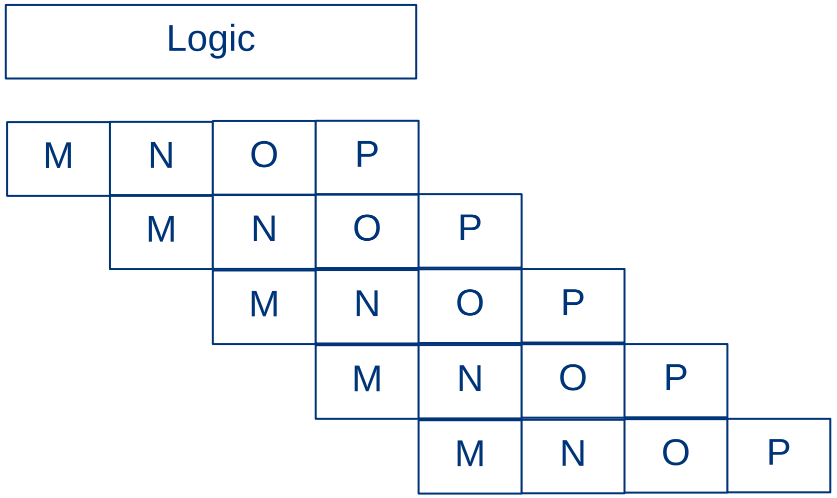
- **4-stage pipeline**



Throughput = 1 / delay of stage

# Theoretical Analysis

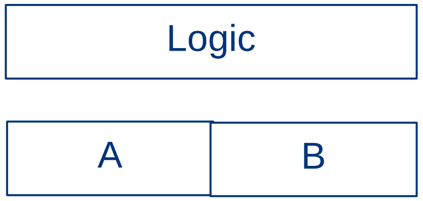
- **4-stage pipeline**



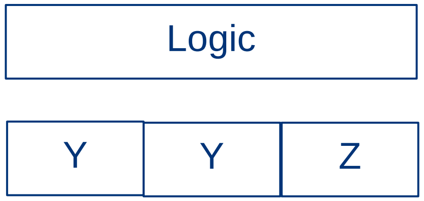
Throughput = 1 / delay of stage

# Delay of a stage

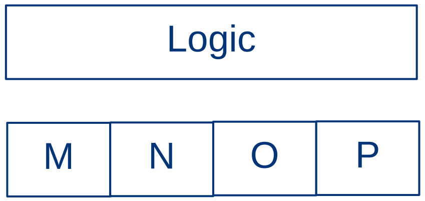
- **2-stage pipeline**



- **3-stage pipeline**

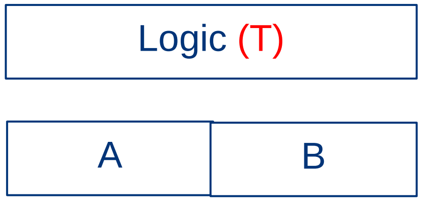


- **4-stage pipeline**



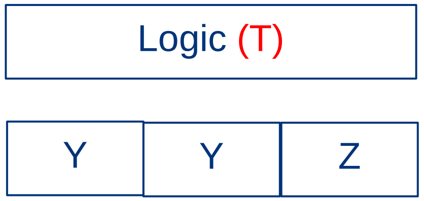
## Delay of a stage

- **2-stage pipeline**



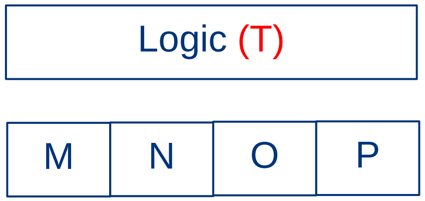
Delay of a stage = T/2

- **3-stage pipeline**



Delay of a stage = T/3

- **4-stage pipeline**

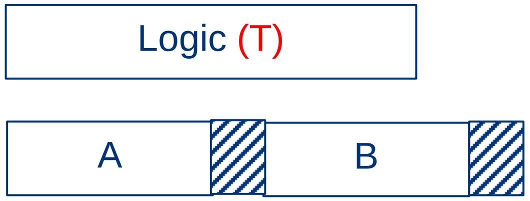


Delay of a stage = T/4

# Delay of a stage

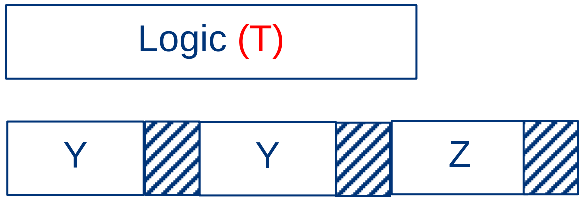
▨ Latch with delay S

- **2-stage pipeline**



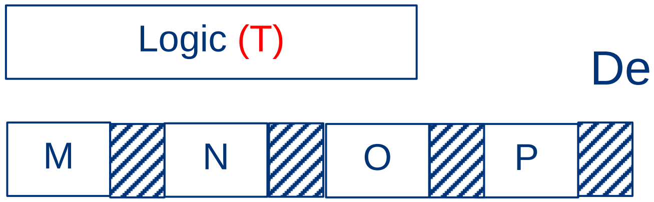
Delay of a stage = T/2 + S

- **3-stage pipeline**



Delay of a stage = T/3 + S

- **4-stage pipeline**

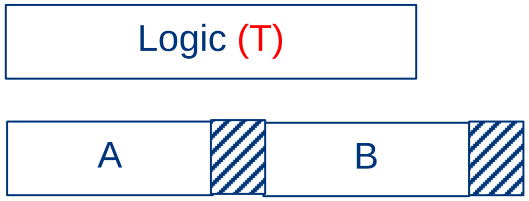


Delay of a stage = T/4 + S

# Delay of a stage

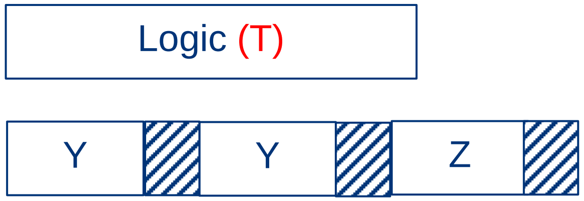
▨ Latch with delay S

- **2-stage pipeline**



Delay of a stage = T/2 + S

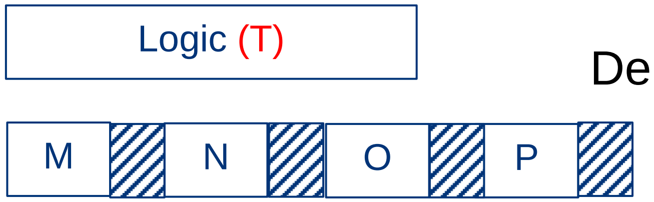
- **3-stage pipeline**



Delay of a stage = T/3 + S

Throughput = 1 / delay

- **4-stage pipeline**

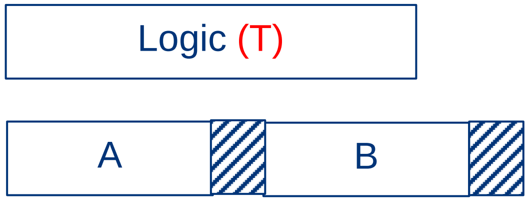


Delay of a stage = T/4 + S

# Delay of a stage

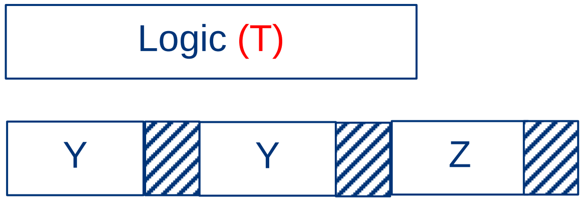
▨ Latch with delay S

- **2-stage pipeline**



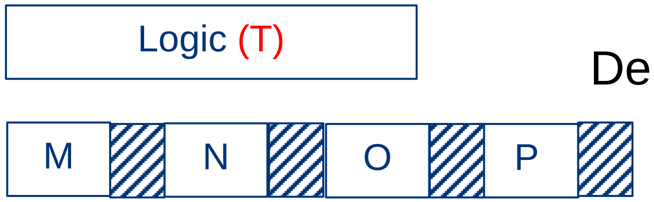
Delay of a stage = T/2 + S   Throughput = 1 / (T/2 + S)

- **3-stage pipeline**



Delay of a stage = T/3 + S   Throughput = 1 / (T/3 + S)

- **4-stage pipeline**

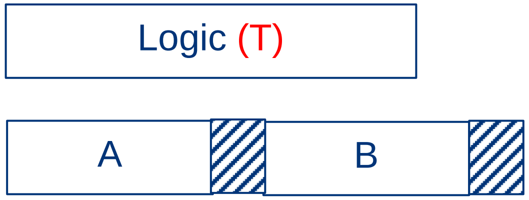


Delay of a stage = T/4 + S   Throughput = 1 / (T/4 + S)

# Delay of a stage

▨ Latch with delay S

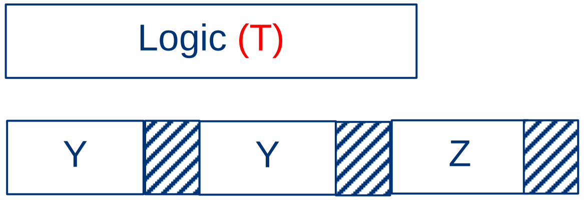
- **2-stage pipeline**



Delay of a stage = T/2 + S    Throughput = 1 / (T/2 + S)

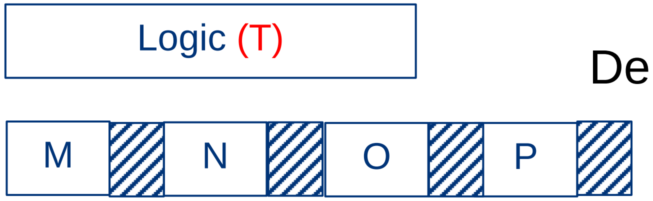
Throughput = 1 / (T/k + S), k = number of stages

- **3-stage pipeline**



Delay of a stage = T/3 + S    Throughput = 1 / (T/3 + S)

- **4-stage pipeline**

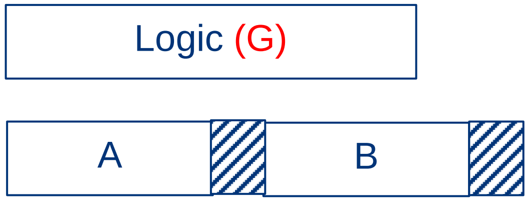


Delay of a stage = T/4 + S  Throughput = 1 / (T/4 + S)

# Area Analysis

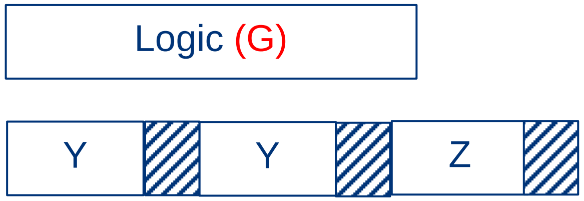
▨ Latch with area L

- **2-stage pipeline**



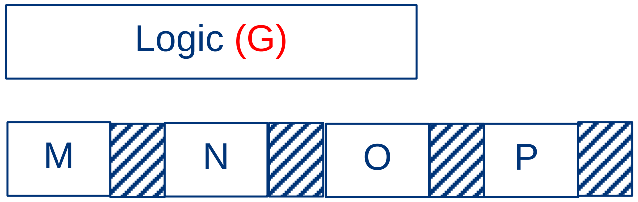
**A = G/2, B = G/2**
**Total area = G/2 + L + G/2 + L = G + 2L**

- **3-stage pipeline**



**X = G/3, Y = G/3, Z = G/3**
**Total area = G/3 + L + G/3 + L + G/3 + L = G + 3L**

- **4-stage pipeline**

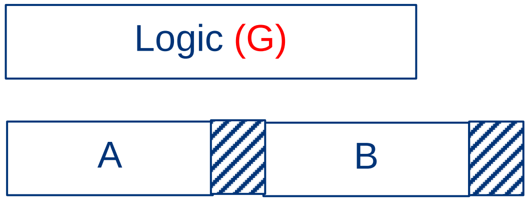


**M = G/4, N = G/4, O = G/4, P = G/4**
**Total area = G + 4L**

# Area Analysis

▨ Latch with area L

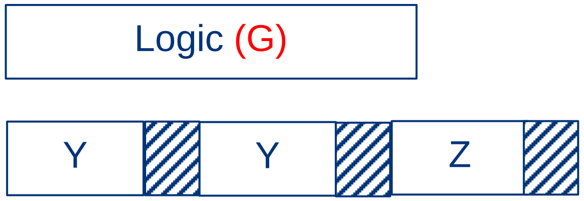
- **2-stage pipeline**



**A = G/2, B = G/2**
**Total area = G/2 + L + G/2 + L = G + 2L**

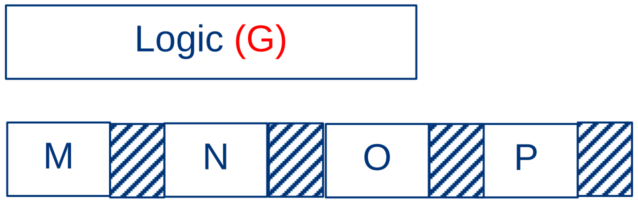
Area = G + k*L, k = number of stages

- **3-stage pipeline**



**X = G/3, Y = G/3, Z = G/3**
**Total area = G/3 + L + G/3 + L + G/3 + L = G + 3L**

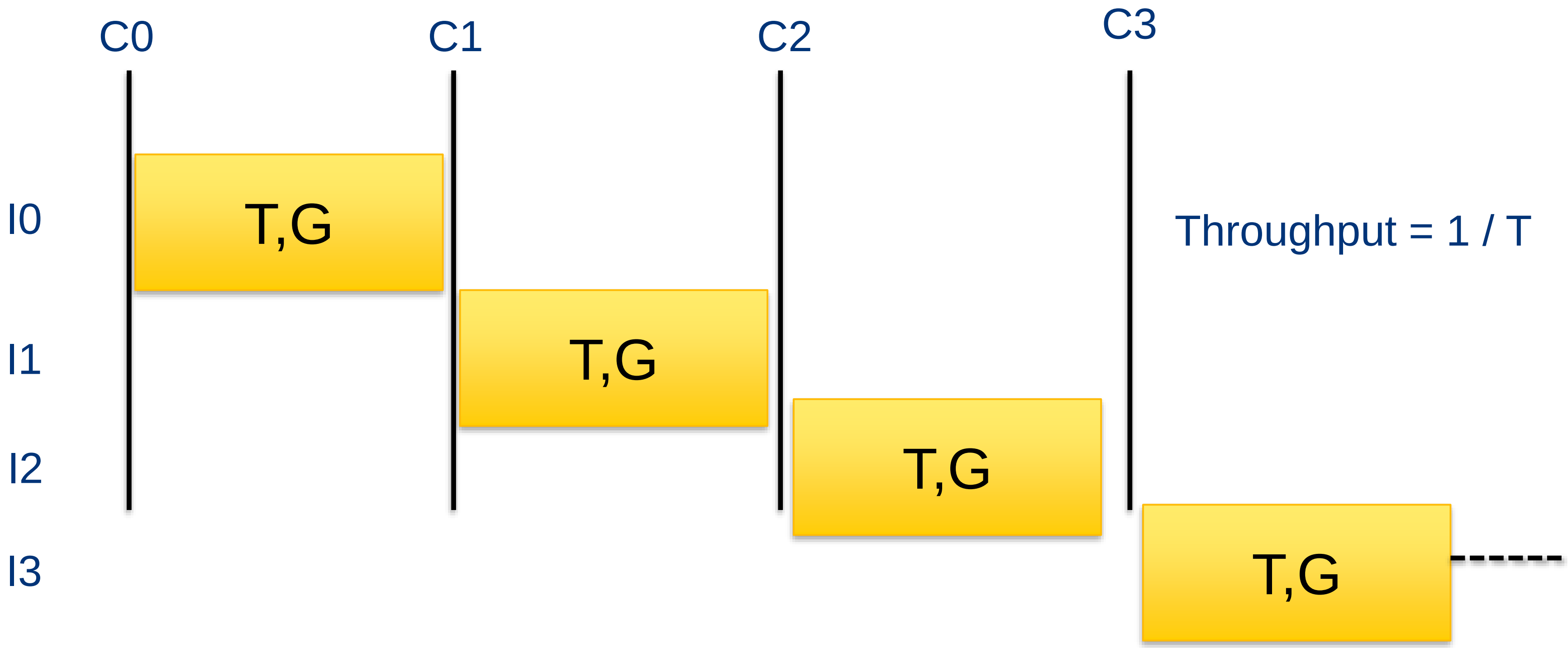
- **4-stage pipeline**



**M = G/4, N = G/4, O = G/4, P = G/4**
**Total area = G + 4L**

# Theoretical Analysis

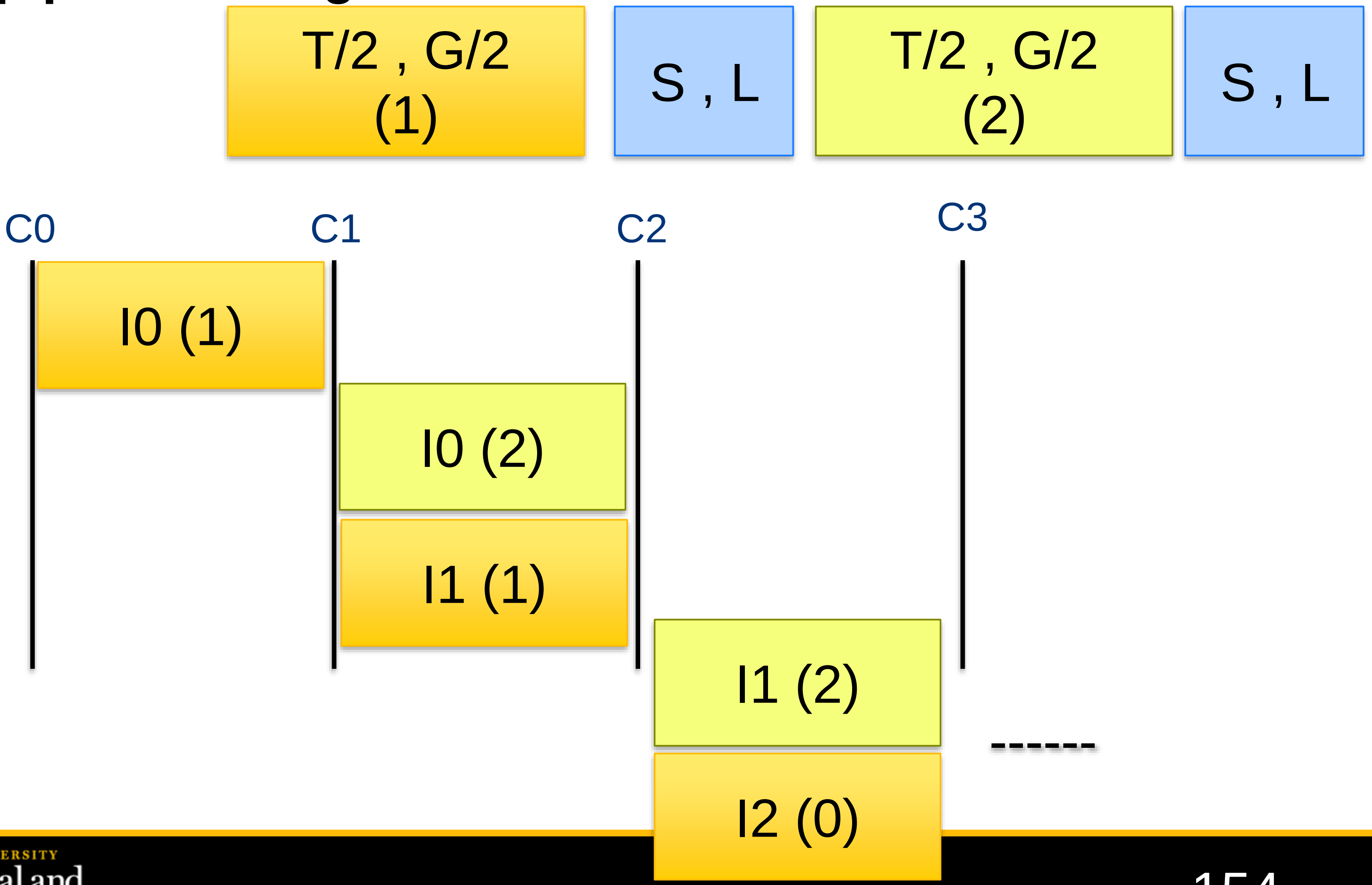
- **Big Combinational Logic**
  - Delay = T ns
  - Hardware = G gates



# Theoretical Analysis

Throughput = 1 / (T/2 + S)

- **Two pipelined stages**

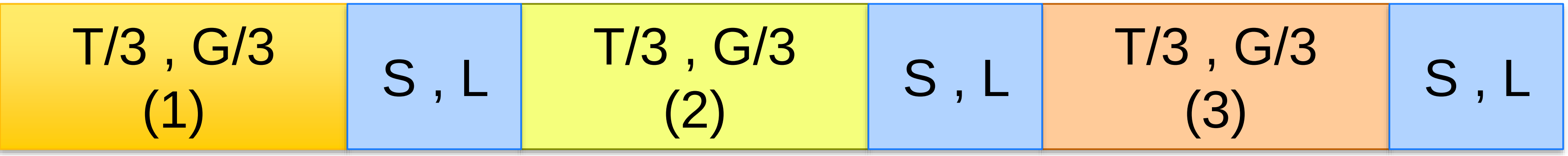


## Theoretical Analysis

- Throughput = 1 / (T/2 + S)
- Area = G/2 + L + G/2 + L = G + 2*L

## Theoretical Analysis

- **For 3 stages**



- **Throughput = 1 / delay of a stage = 1 / (T/3 + S)**
- **Total Area = G/3 + L + G/3 + L + G/3 + L = G + 3*L**

## Theoretical Analysis

- k-stages pipeline
- Throughput = 1 / delay of a stage = 1 / (T/k + S)
- Area = G + k*L

## More Realistic Pipeline: Throughput and Cost

- **$TP_{k\text{-stage}}$ = 1 / (T/k +S )**
  - Objective is to maximize throughput
- **$Cost_{k\text{-stage}}$ = G + Lk**
  - Objective is to minimize cost

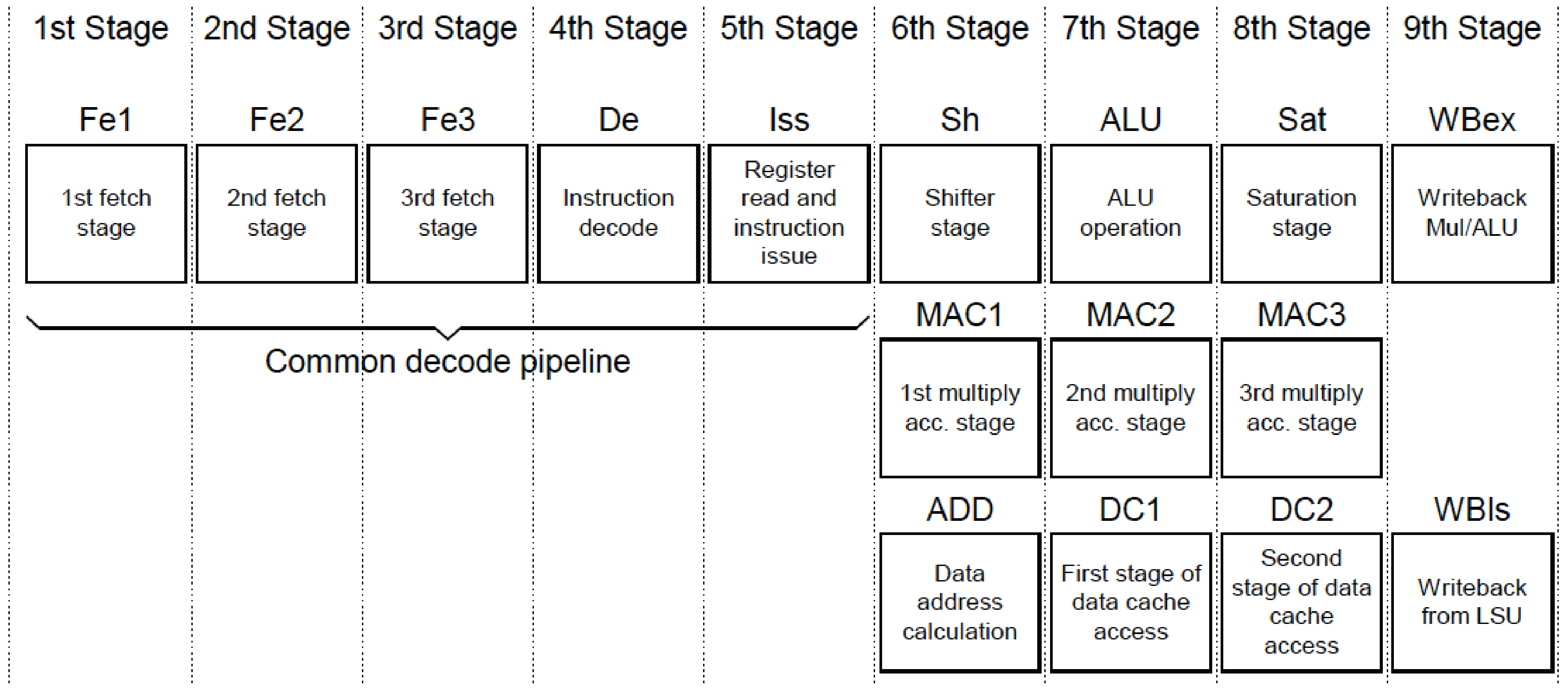
$$\frac{\text{Cost}}{\text{Perf}} = \frac{kL+G}{\frac{1}{\frac{T}{k}+S}} = (kL+G)\left(\frac{T}{k}+S\right) =$$

$$LT + GS + LSk + \frac{GT}{k}$$

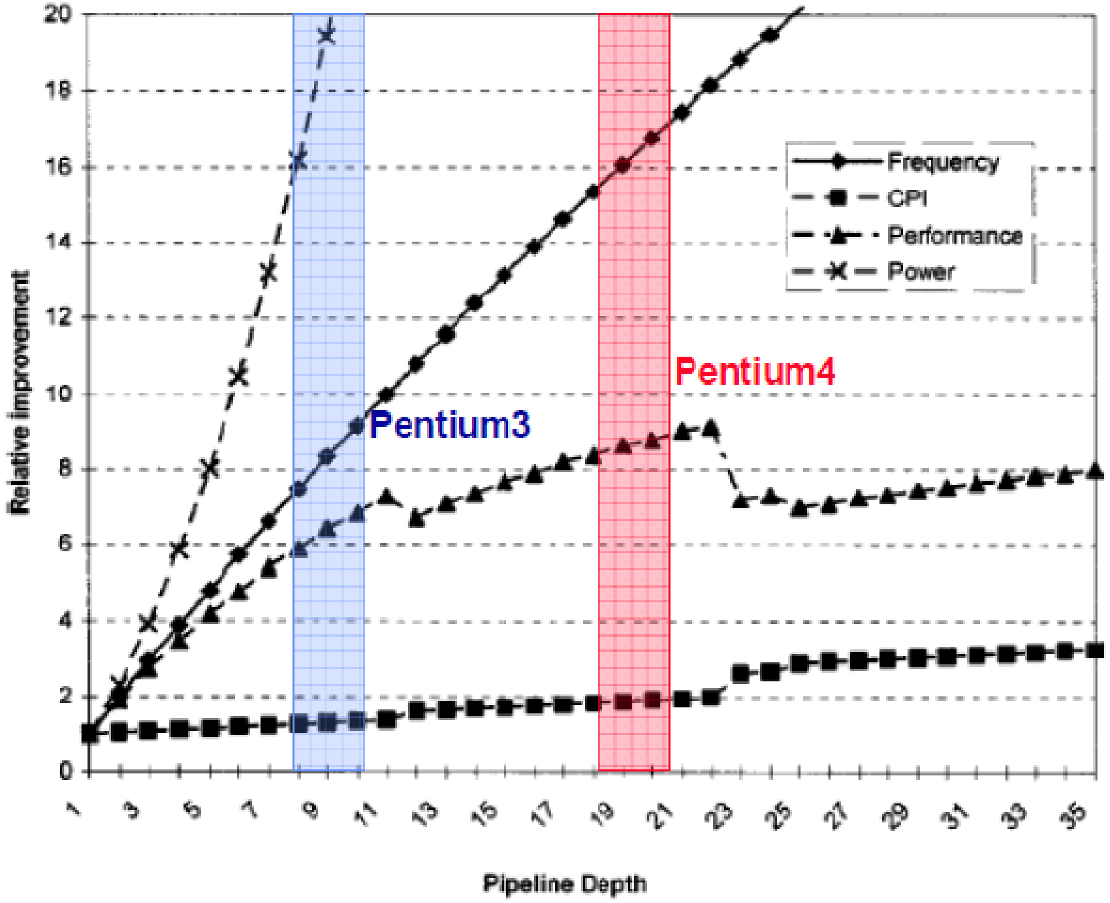
$$\frac{\mathrm{d}}{\mathrm{d}k}\left(\frac{\text{Cost}}{\text{Perf}}\right) = 0 + 0 + LS - \frac{GT}{k^2}$$

$$LS - \frac{GT}{k^2} = 0 \Rightarrow k_{\text{opt}} = \sqrt{\frac{GT}{LS}}$$

# ARM Pipeline Stages



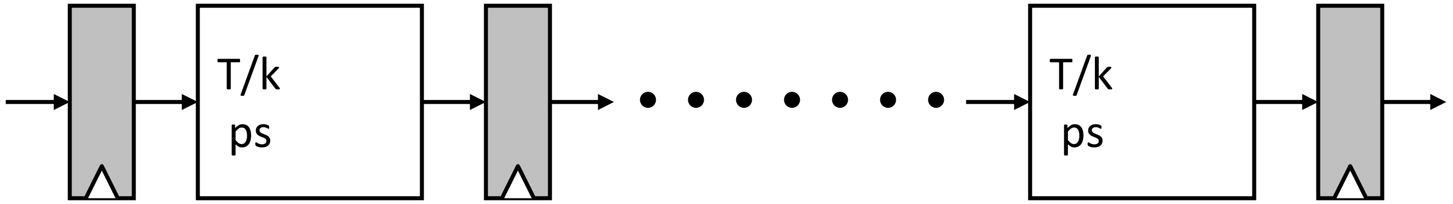
## Perspective on Deeper Pipeline



Source: Ed Grochowski, 1997

# Pitfall

- **$TP_{k\text{-stage}} = 1 / (T/k + S)$**
  - We have assumed that the stages are all balanced
  - What if they are not?
    - We will see how the throughput deviates from ideal throughput for imbalanced pipeline
  - Throughput computation: 1 / (maximum delay of all stages)



## Pipeline Design Steps

- **Balancing work in pipeline stages**
- **Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow**
  - Dependencies
  - Resource contentions
  - Long latency operations
- **Handling exceptions, interrupts**
- **Improving pipeline throughput**

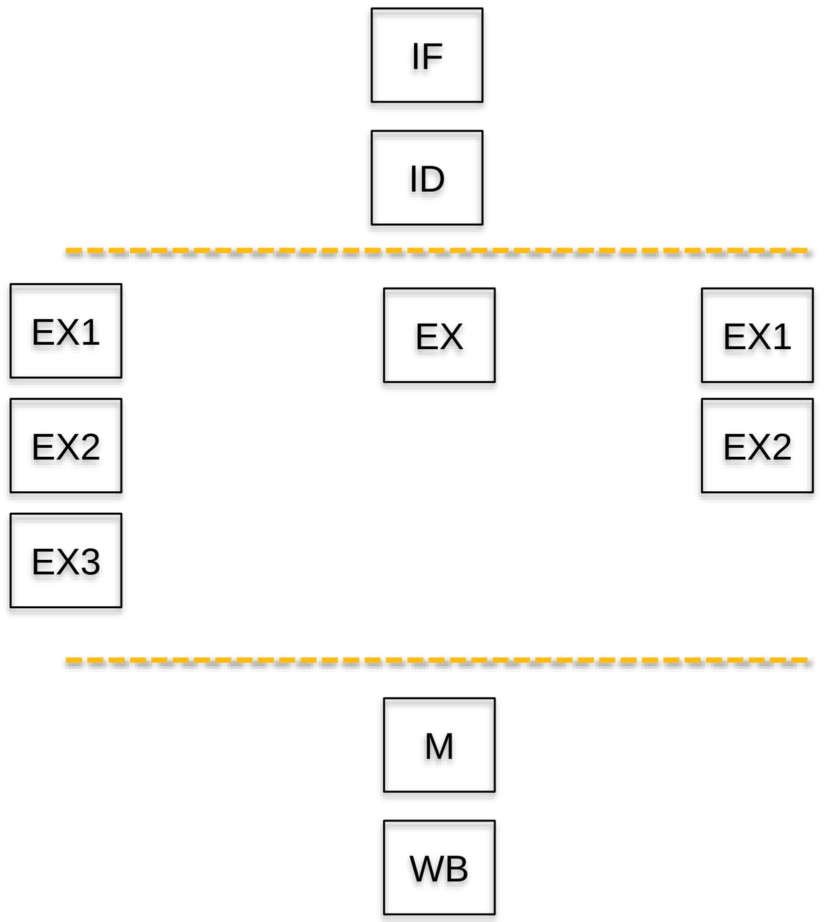
## Pipeline Design Steps

- Balancing work in pipeline stages
- **Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow**
  - Dependencies
  - Resource contentions
  - Long latency operations
- **Handling exceptions, interrupts**
- **Improving pipeline throughput**

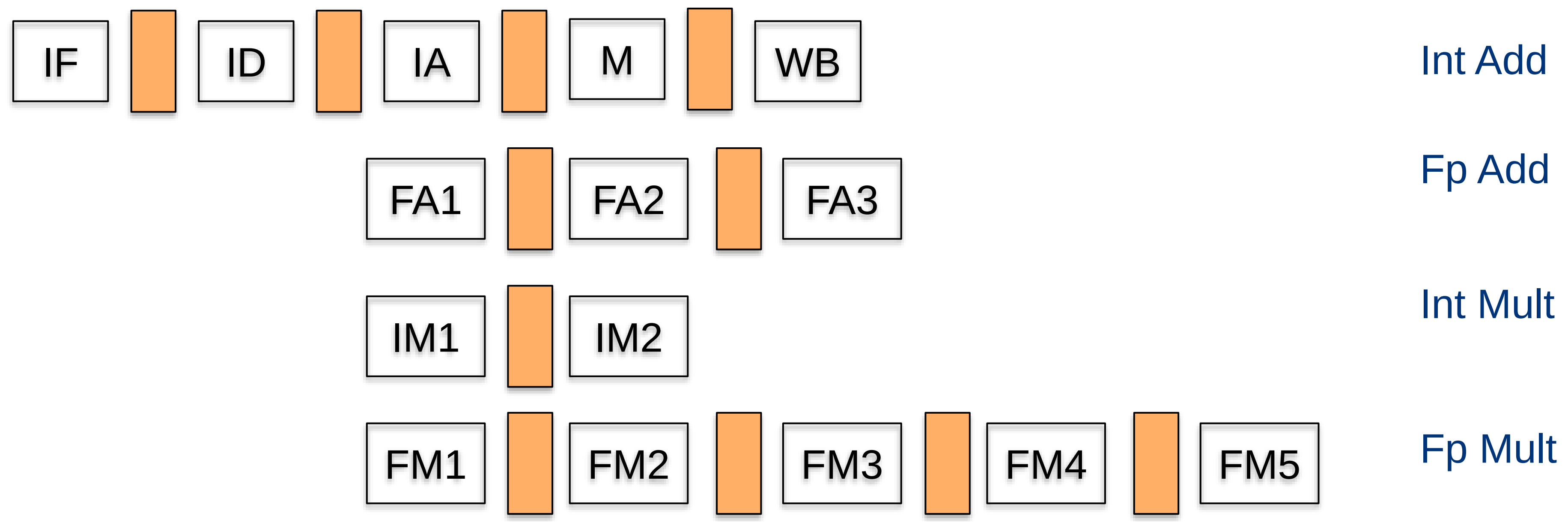
# Balancing Pipeline Stages

- **Divide a big operation into small sub-operations**
- **Merge sub-operations**
- **Deeper pipelines**
- **Pipelining of memory accesses**
- **Multiple different pipelines or sub-pipelines**
  - Pipelining of long latency operations

# Modern Architecture



# Running Pipeline Example



## Pipeline Design Steps

- Balancing work in pipeline stages

- **Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow**
  - Dependencies
  - Resource contentions
  - Long latency operations
- **Handling exceptions, interrupts**

- **Improving pipeline throughput**