# Instruction-Level Parallelism and Its Dynamic Exploitation

陈文智

chenwz@zju.edu.cn
浙江大学计算机学院
2014年9月

# 内容提要及各节间的关系(1)

3.1 流水线技术基础

3.2 How Is Pipelining Implemented?

**3.3** The Major Hurdle of Pipelining—Pipeline Hazards

3.4 Extending the MIPS Pipeline to Handle Multicycle Operations

*(本科回顾 Appendix A)*

- 流水线技术就是指令重叠执行技术，达到加快运算速度的目的
- 由于存在三种流水线竞争：结构竞争、数据竞争、控制竞争，导致流水线性能降低，不能运作在理想的重叠状态，需要插入停顿周期，从而使流水线性能降低。

# 内容提要及各节间的关系(2)

3.5 Instruction-Level Parallelism: Concepts and Challenges

(CPI=1)

- 指令之间可重叠执行性称为指令级并行性（Instruction Parallelism-ILP)。因此进一步研究和开发指令之间的并行性，等于拓宽指令重叠执行的可能性，从而能进一步提高流水线的性能。

# 内容提要及各节间的关系（3）

3.6 Overcoming Data Hazards with Dynamic Scheduling (2.4)

- 针对流水线数据竞争的动态调度

3.7 Reducing Branch Costs with Dynamic Hardware Prediction (2.3)

- 针对流水线控制竞争的预测技术

# 内容提要及各节间的关系(4)

3.8 Hardware-Based Speculation (2.6)

- 进一步开发指令级并行性的动态技术
  - 跨控制流的动态调度:数据竞争+控制竞争

3.9 Taking Advantage of More ILP with Multiple Issue(2.7)

- 进一步开发指令级并行性(CPI<1)
  - 采用单位时钟发射多条指令

# 3. 1流水线技术基础

*本科回顾------- Appendix A.1*

3.1.1 What is pipelining?

3.1.2 Why pipelining ?

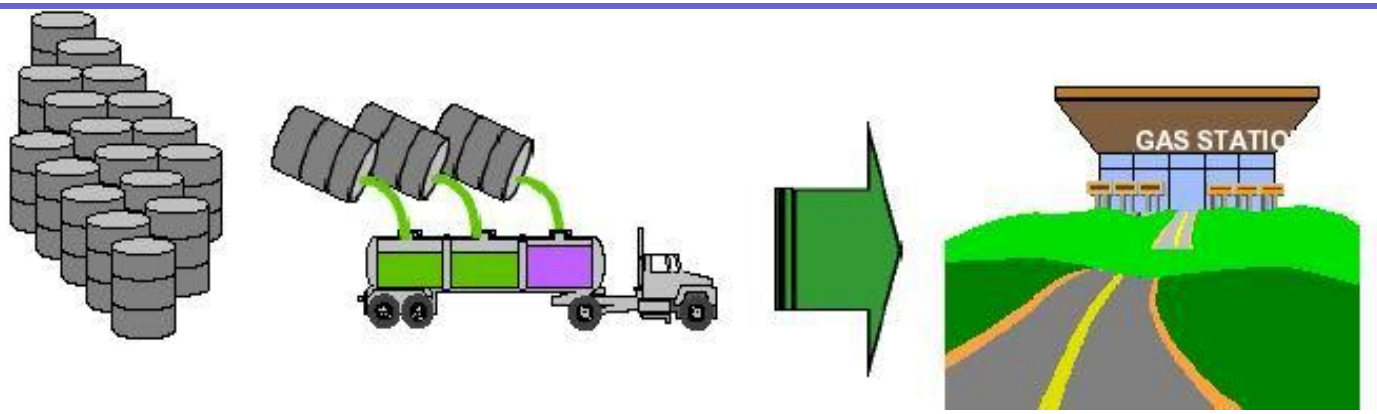
3.1.3 Ideal Performance for Pipelining

# 3.1.1 What is pipelining?

- Pipelining:
  - *“A technique designed into some computers to increase speed by starting the execution of one instruction before completing the previous one.”*

    *----Modern English-Chinese Dictionary*
  - implementation technique whereby different instructions are overlapped in execution at the same time.
  - implementation technique to make fast CPUs

# Trucking gas from depot to gas station

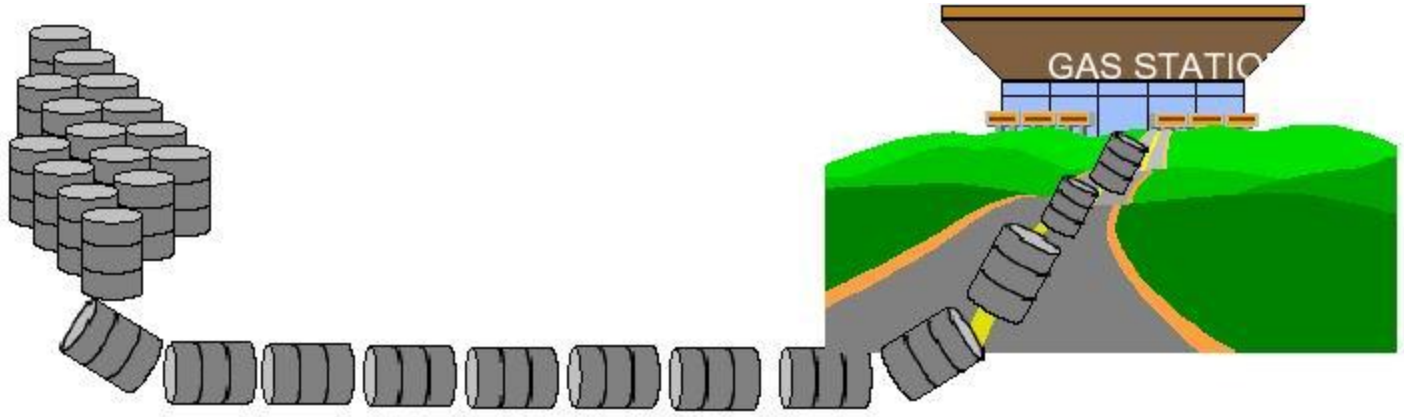



- The steps:
  - Get the barrels
  - Load them into the truck
  - Drive to the gas station
  - Unload the gas
  - Return for more oil
- Let's do the math
  - Each truck can carry 5 barrels
  - Can load a truck with 5 barrels in 1 hour
  - It takes each truck 1 day to drive to and from gas station
  - How many barrels per week are delivered?

# Like a Multi-cycle Processor

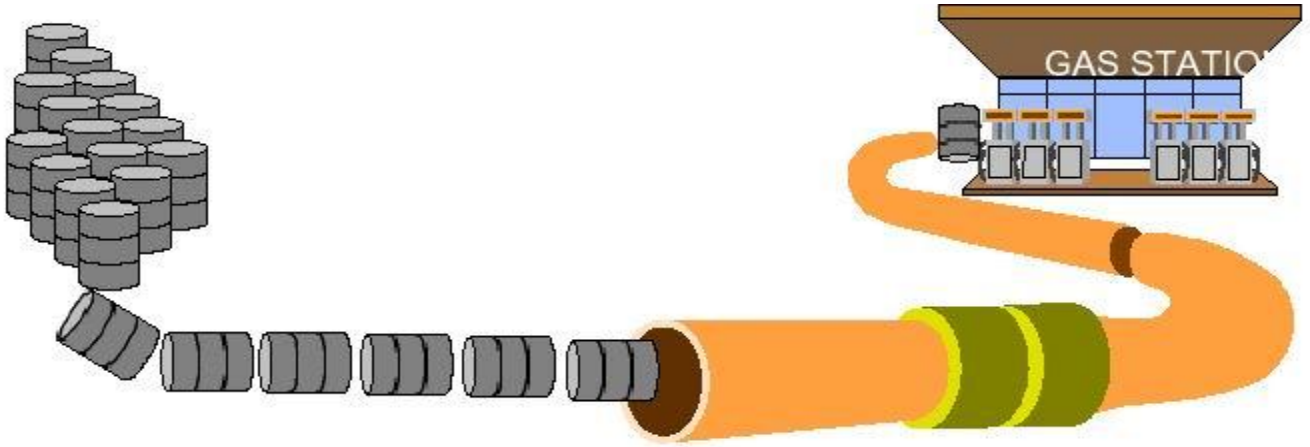
- What are similar in the steps ?
  - Fetch an instruction (Get the barrels)
  - Decode the instruction (Load them into the truck)
  - ALU OP (Drive to the gas station)
  - Memory Access (Unload the gas)
  - Write-back (Return for more oil)

# A better way, but dangerous



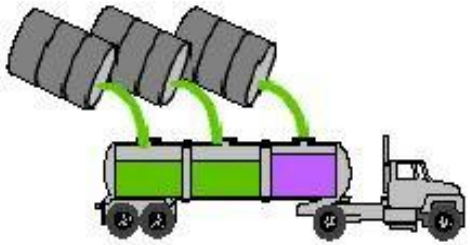

- Roll the barrels down the road
  - Big fire hazard

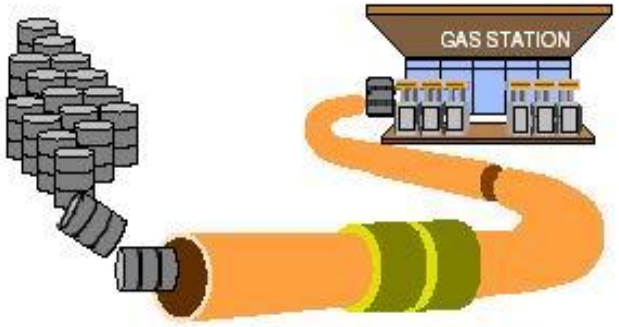
# Big idea: Build a pipeline




- Now let's do the math
  - Pipeline can accept 1 barrel every hour
  - How many barrels get delivered to the gas station per day?

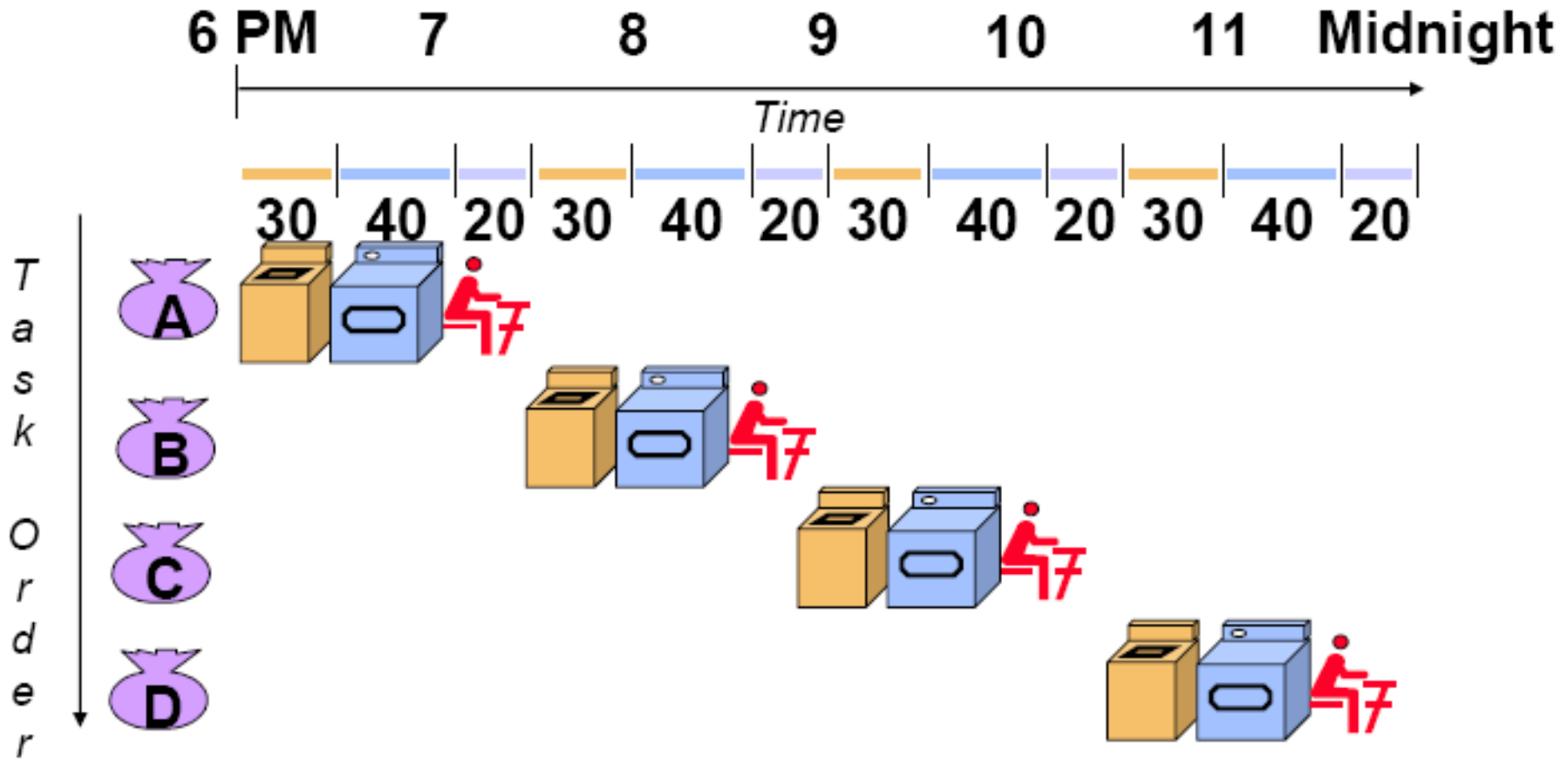
# Trucking vs. Pipelines





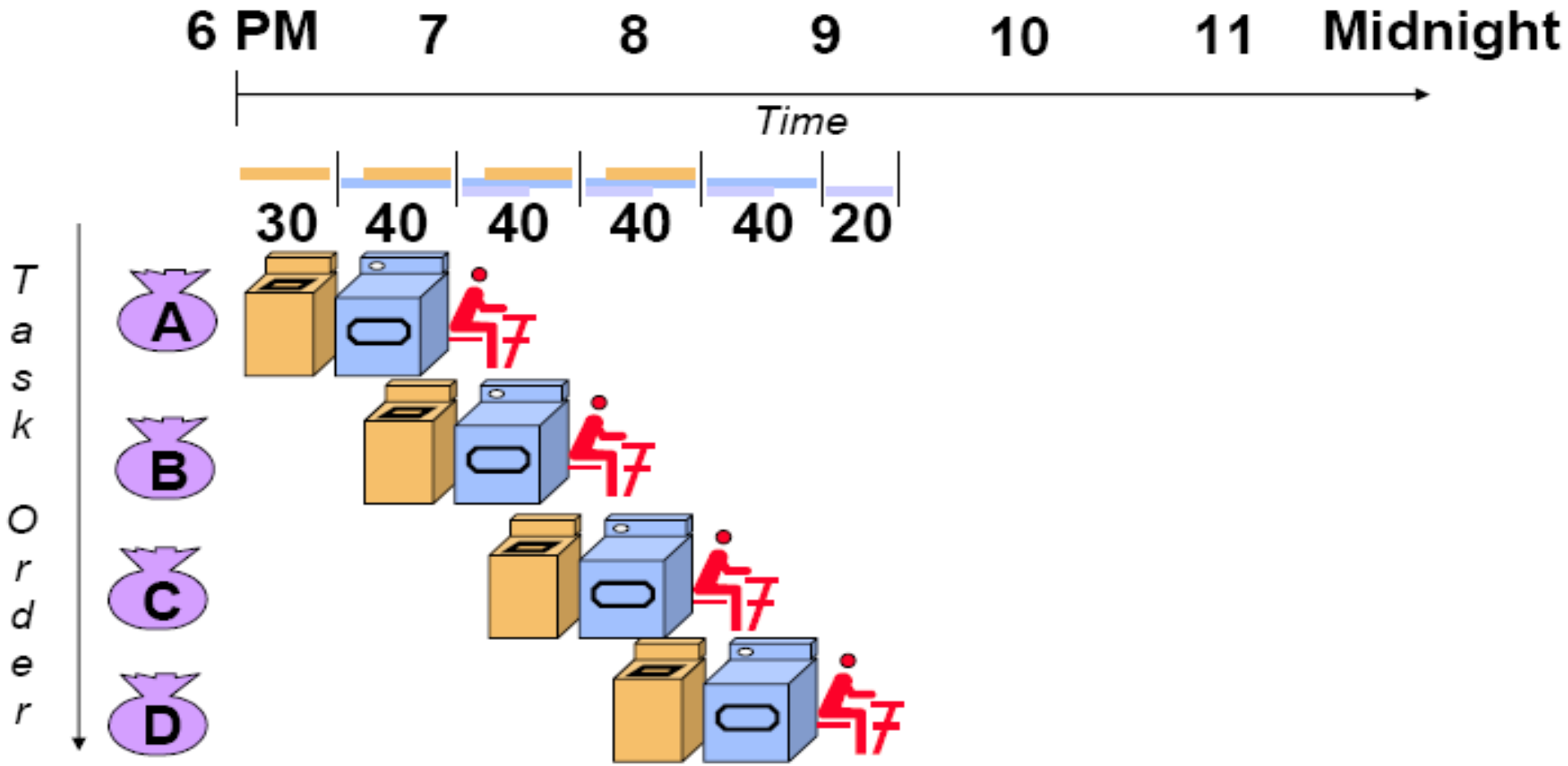

- Trucks
  - Truck with 5 barrels takes 1 day to drive to and from gas station, while need 2 hours for loading and unloading
  - LOTS of TIME when loading area,gas station, and pieces of the road are unused
- Pipelines
  - Pipeline can accept 1 barrel every hour
  - Resources (loading area, gas station,pipelines) are always in use

# Sequential Laundry



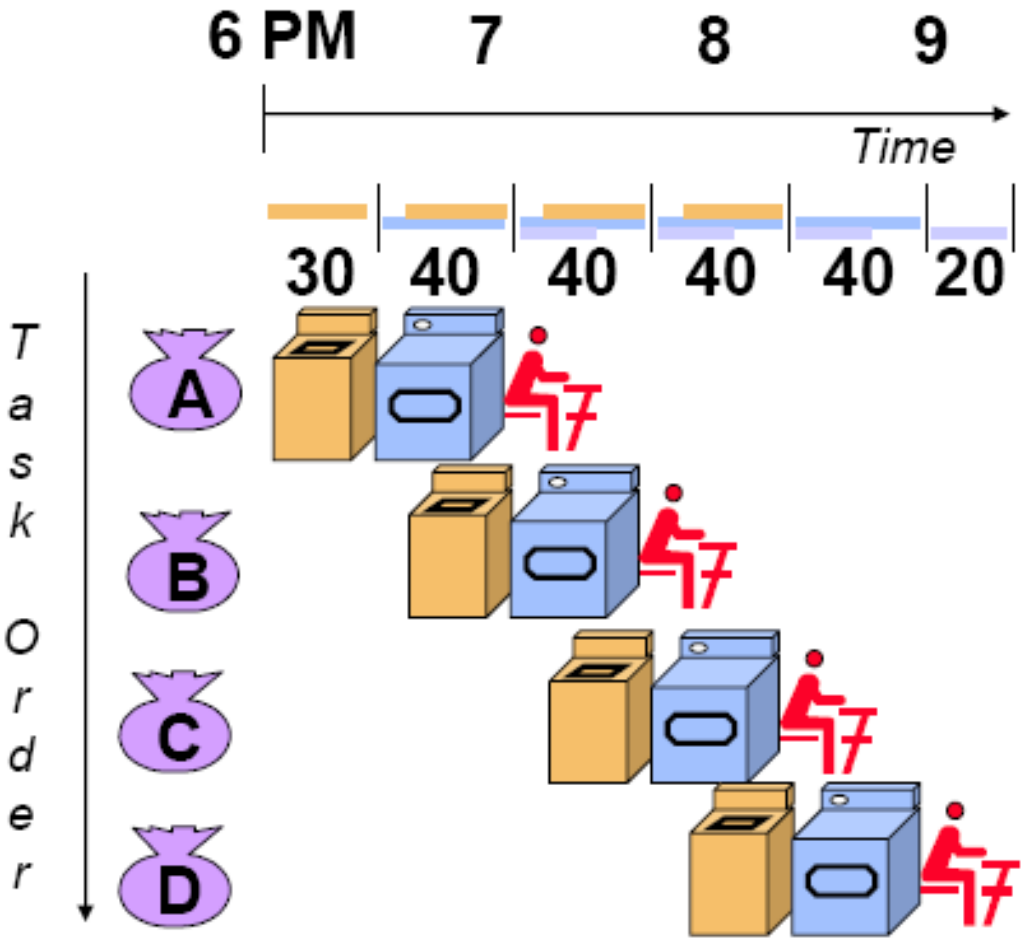

- Washer takes 30 min, Dryer takes 40 min, folding takes 20 min
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

# Pipelined Laundry




- Pipelining means start work as soon as possible
- Pipelined laundry takes 3.5 hours for 4 loads

# Pipelining Lessons

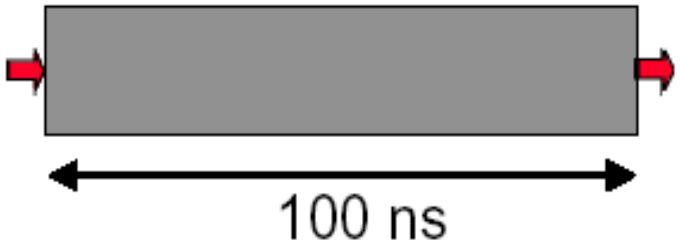



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduce speedup
- Stall for Dependencies

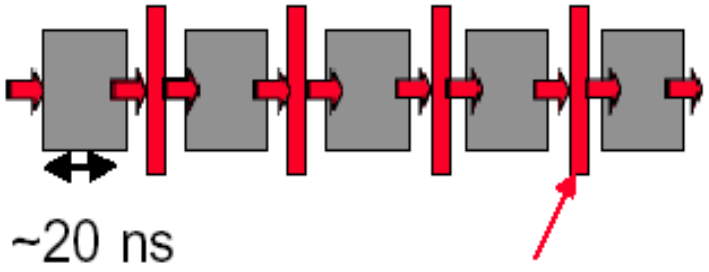
# What is Pipelining

- A pipeline is like an auto assemble line
- A pipeline has many stages
- Each stage carries out a different part of instruction or operation
- The stages, which cooperates at a synchronized clock, are connected to form a pipe
- An instruction or operation enters through one end and progresses through the stages and exit through the other end
- Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream

# 3.1.2 Why pipelining : save time and high utilization factor

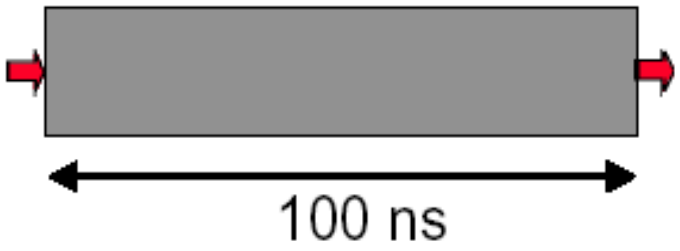



- Only deal one task each time.
- This task takes " such a long time"

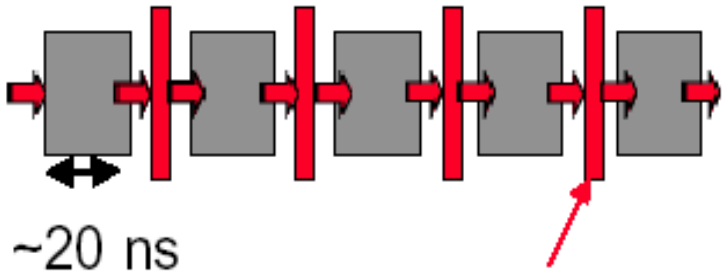



- Latches, called pipeline registers' break up computation into 5 stages
- Deal 5 tasks at the same time.

# Why pipelining: How faster




- Can "launch" a new computation every 100ns in this structure
- Can finish $10^7$ computations per second




- Can launch a new computation every 20ns in pipelined structure
- Can finish $5 \times 10^7$ computations per second

# Conclusion

- The key implementation technique used to Make fast CPU: decrease CPUtime.
- Improving of Throughput ( rather than individual execution time)
- Improving of efficiency for resources (functional unit)

# 3.1.3 Ideal Performance for Pipelining

$$Speedup = \frac{\text{Time tasks on unpipelined machine}}{\text{Time same tasks on pipelined machine}}$$

Assume: stages: k tasks: n

$$T_k = (k+(n-1))\tau_p$$

$$T_1 = n\,k\tau_{up}$$

$$S_{peedup} = \frac{T_1}{T_k} = \frac{nk\tau_{up}}{k\tau_p + (n-1)\tau_p}$$

$n \rightarrow \infty$ $S_{peedup} \rightarrow K$

# Ideal Performance for Pipelining

- If the stages are perfectly balanced, The time per instruction on the pipelined processor equal to:

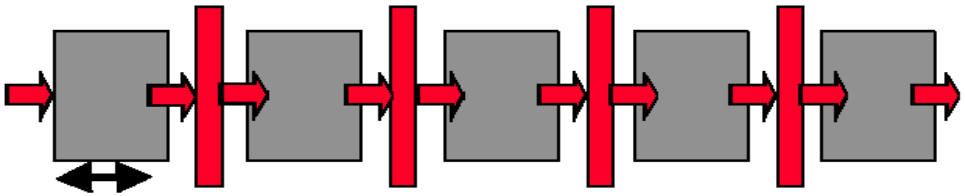
$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

- So, **Ideal speedup equal to**

  **Number of pipe stages.**

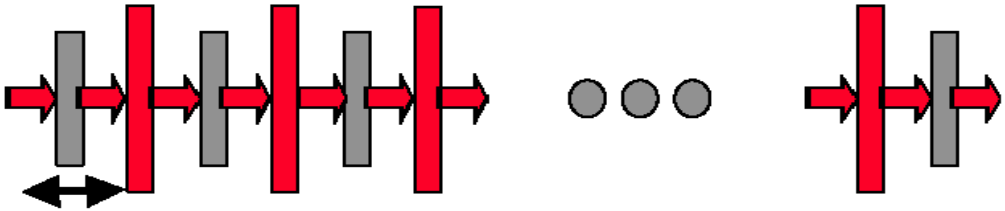
# Why not just make a 50-stage pipeline ?

- Some computations just won't divide into any finer (shorter in time) logical implementation.



~20 ns

**5 stages OK**

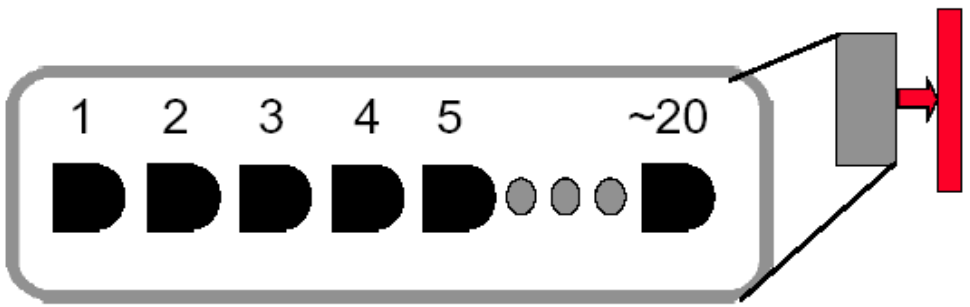


~2 ns

**50 stages NO. Sorry!**

# Why not just make a 50-stage pipeline ?

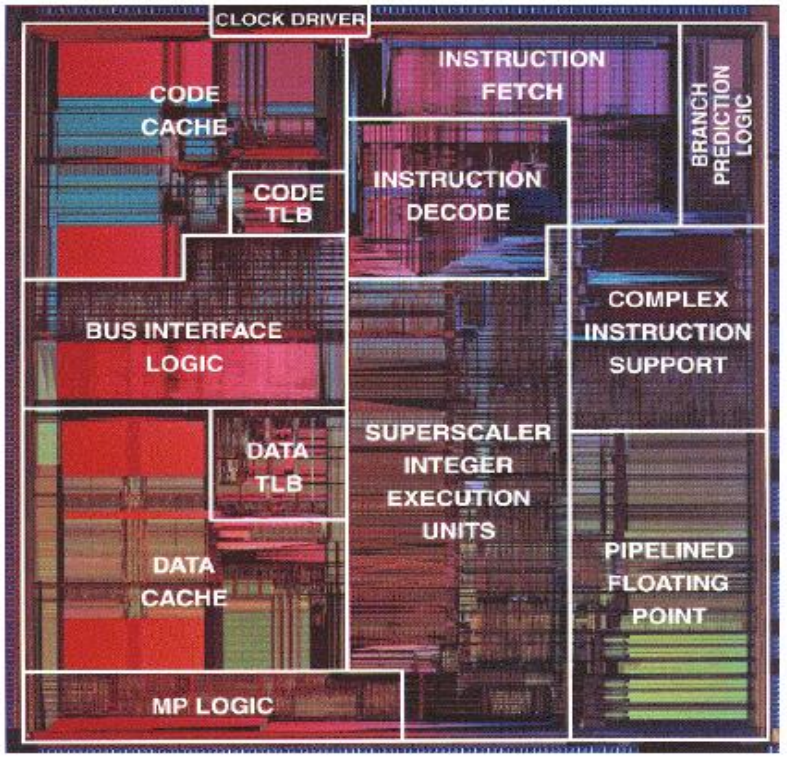
- Those latches are **NOT** free, they take up **area**, and there is a real **delay** to go THRU the latch itself.
  - Machine cycle > latch latency + clock skew
- In modern, deep pipeline (10-20 stages), this is a real effect
- Typically see logic "depths" in one pipe stage of 10-20 "gates".




At these speeds, and with this few levels of logic, latch delay is important

# How Many Pipeline Stages?

- E.g., Intel
  - Pentium III, Pentium 4: 20+ stages
  - More than 20 instructions in flight
  - High clock frequency (>1GHz)
  - High IPC
- Too many stages:
  - Lots of complications
  - Should take care of possible dependencies among in-flight instructions
  - Control logic is huge



# 3.2 How Is Pipelining Implemented?

*本科回顾------- Appendix A.3*

3.2.1 How does instruction Work in the MIPS 5 stage pipeline?

3.2.2 5-stage Version of MIPS Datapath

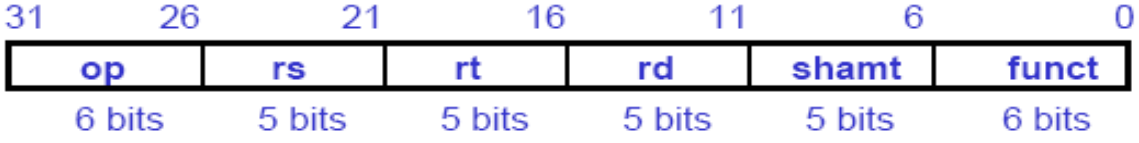
3.2.3 The MIPS pipelining and some Problems

# Basic of RISC Instruction Set

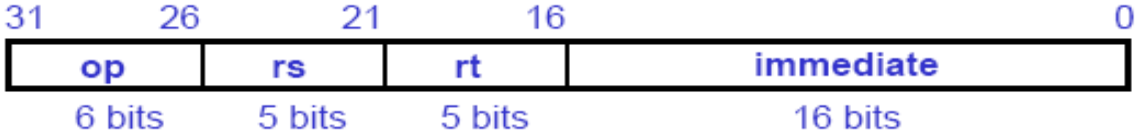
- RISC architectures are characterized by the following features that dramatically simplifies the implementation:
  1. All ALU operations apply only on data in registers
  2. Memory is affected only by load and store operations
  3. Instructions follow very few formats and typically are of the same size

- All MIPS instructions are 32 bits, following one of three formats:

R-type




I-type




J-type

# MIPS Instruction Format

## ❶ Register-format instructions:

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

*op*: Basic operation of the instruction, traditionally called opcode
*rs*: The first register source operand
*rt*: The second register source operand
*rd*: The register destination operand, it gets the result of the operation
*shmat*: Shift amount (explained in future lectures)
*funct*: This field selects the specific variant of the operation of the op field

❑ MIPS assembly language includes two conditional branching instructions using PC -relative addressing:

```
beq register1, register2, L1   # go to L1 if (register1) = (register2)
bne register1, register2, L1   # go to L1 if (register1) ≠ (register2)
```

❑ Examples:

```
add    $t2, $ t1, $ t1    # Temp reg $t2 = 2 $t1
sub    $t1, $s3, $s4      # Temp reg $t1 = $s3 - $s4
and    $t1, $ t2, $ t3    # Temp reg $t1 = $t2 . $t
bne    $s3. $s4. Else     # if $s3 ≠ $s4 jump to Else
```

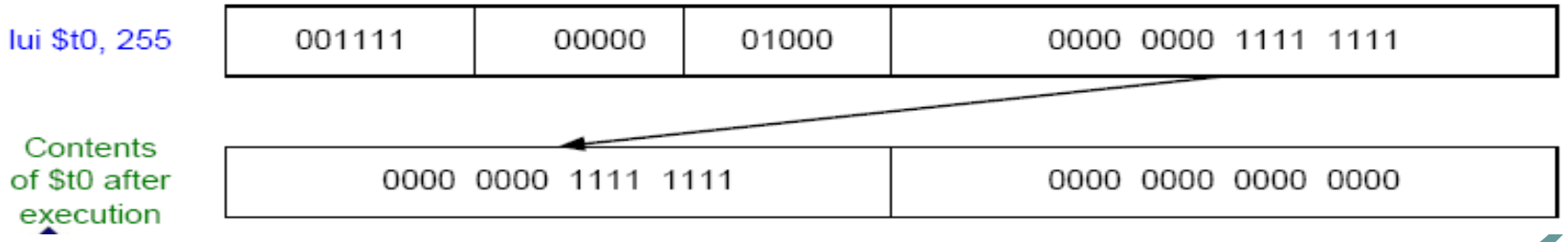
# MIPS Instruction Format

## ❷ Immediate-type instructions:

| op | rs | rt | address |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

- The 16-bit address means a load word instruction can load a word within a region of $\pm 2^{15}$ bytes of the address in the base register
- Examples: lw $t0, 32($s3) , sw $t1, 128($s3)
- MIPS handle 16-bit constant efficiently by including the constant value in the address field of an I-type instruction (Immediate-type)

  addi $sp, $sp, 4 #$sp = $sp + 4
- For large constants that need more than 16 bits, a load upper-immediate (*lui*) instruction is used to concatenate the second part



# Addressing in Jumps and Branches

- I-type instructions leaves only 16 bits for address reference limiting the size of the jump
- MIPS branch instructions use the address as an increment to the PC allowing the program to be as large as $2^{32}$ (called *PC-relative addressing*)
- Since the program counter gets incremented prior to instruction execution, the branch address is actually relative to (PC + 4)
- MIPS also supports an J-type instruction format for large jump instructions

| op | address |
|---|---|
| 6 bits | 26 bits |

- The 26-bit address in a J-type instruct. is concatenated to upper 8 bits of PC

```
Loop:   add   $t1, $s3, $s3
        add   $t1, $t1, $t1
        add    $t1, $t1, $s6
        lw     $t0, 0($t1)
        bne    $t0, $s5, Exit
        add    $s3, $s3, $s4
        j      Loop
Exit:
```

| | | | | | | |
|---|---|---|---|---|---|---|
| 80000 | 0 | 19 | 19 | 9 | 0 | 32 |
| 80004 | 0 | 9 | 9 | 9 | 0 | 32 |
| 80008 | 0 | 9 | 22 | 9 | 0 | 32 |
| 80012 | 35 | 9 | 8 | 0 | | |
| 80016 | 5 | 8 | 21 | 8 | | |
| 80020 | 0 | 19 | 20 | 19 | 0 | 32 |
| 80024 | 2 | 80000 | | | | |
| 80028 | . . . | | | | | |
| 80012 | 35 | 9 | 8 | 0 | | |

# 3.2.1 MIPS 5 stage pipeline (1)

The first two stages of MIPS pipeline

- IF (Instruction fetch cycle)
  - IR←Mem[PC];
  - NPC ←PC=PC+4;
- ID (Instruction decode/register fetch cycle)
  - A ←Regs[rs];
  - B ←Regs[rt];
  - Imm ←sign-extended immediate field of IR;

- Note: The first two stages of MIPS pipeline do the same functions for all kinds of instructions.

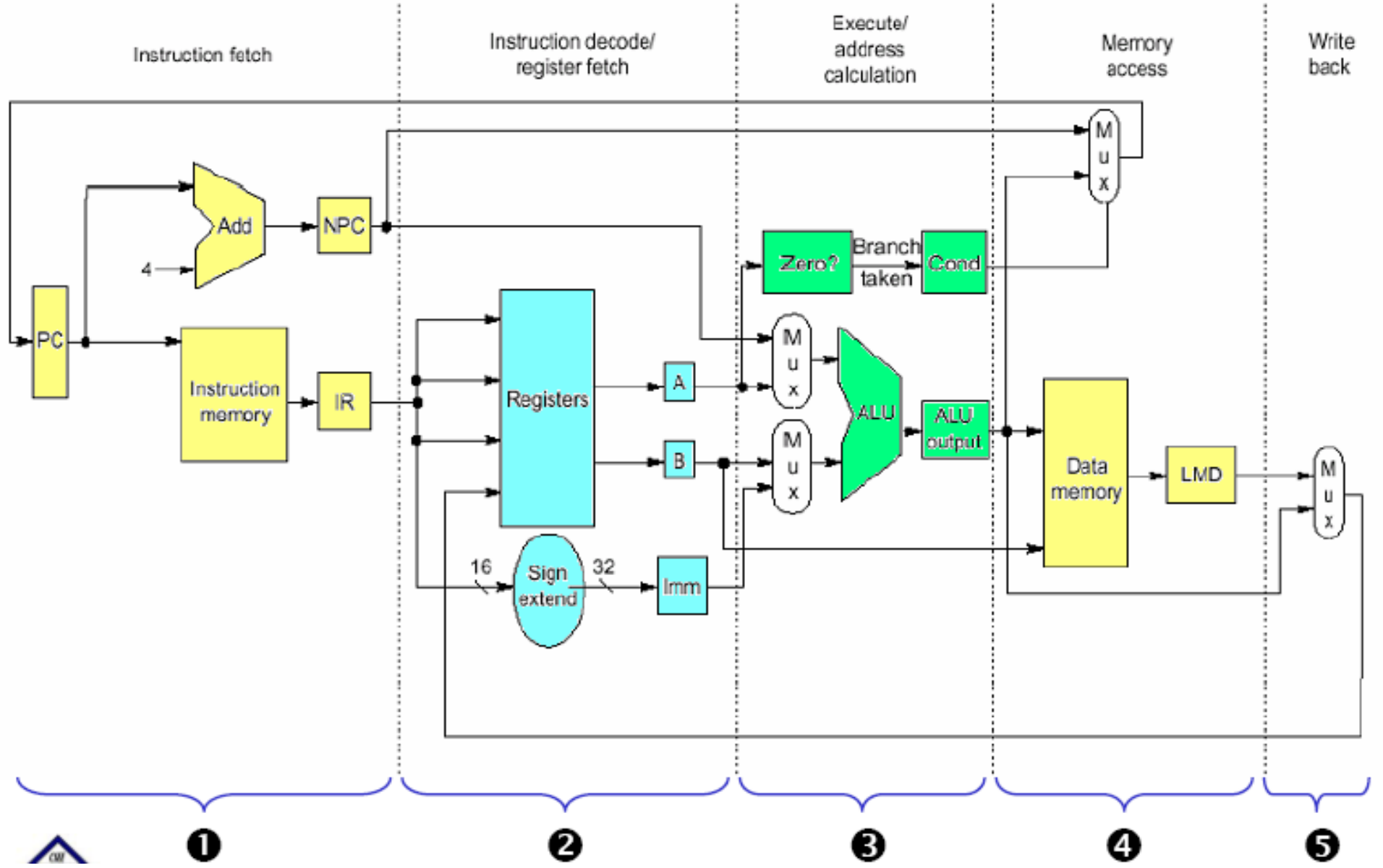
# MIPS 5 stage pipeline (2)

- EX (Execution/effective address cycle)
  - Memory reference:
    - ALUoutput ←A+Imm
  - Register-Register ALU instruction:
    - ALUoutput ←A func B;
  - Register-Immediate ALU instruction:
    - ALUoutput ←A op Imm;
  - Branch:
    - ALUoutput ←NPC+(Imm <<2 );
    - Cond ←(A==0)

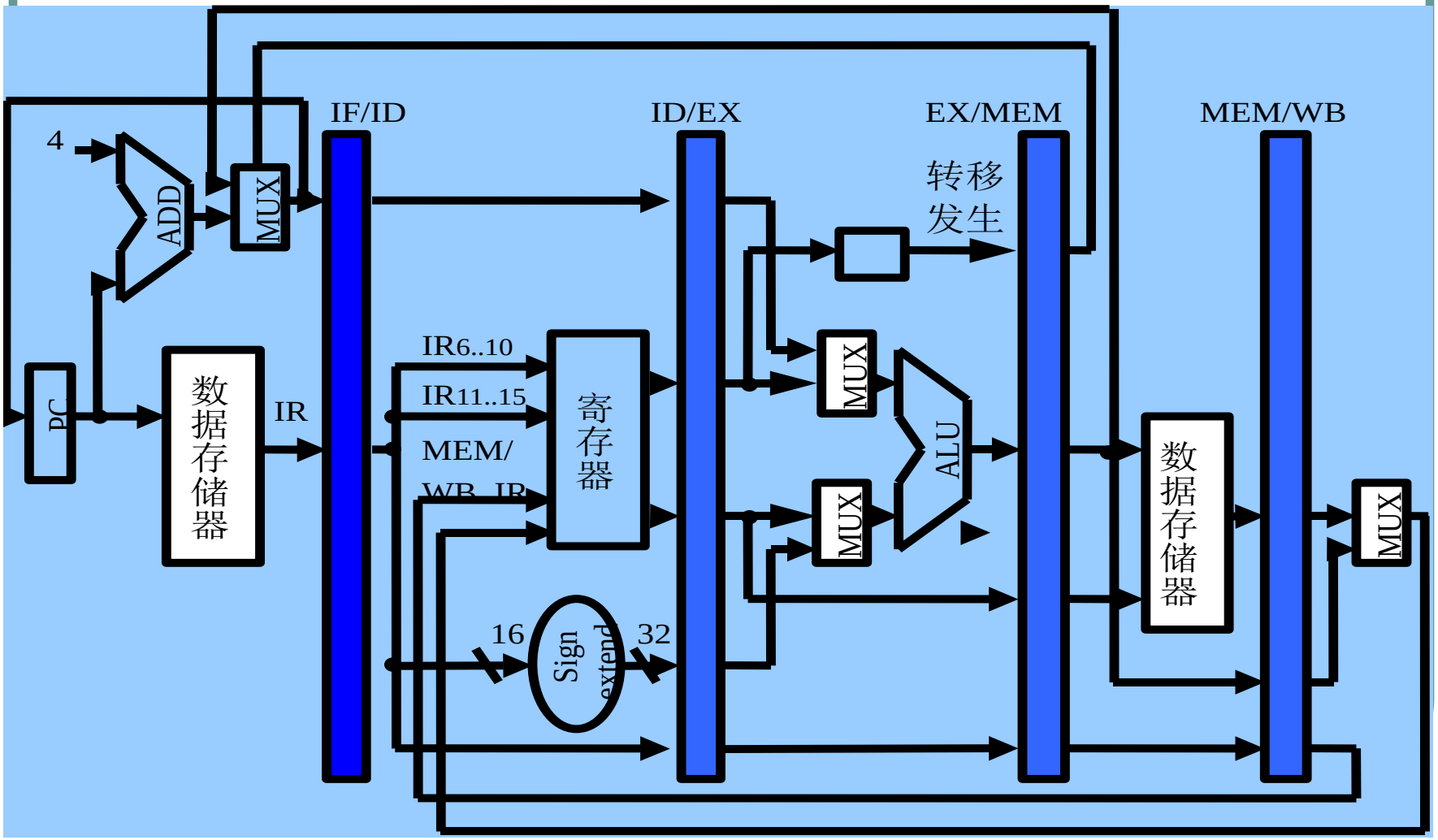
# MIPS 5 stage pipeline (2)

- MEM(Memory acces/branch completion cycle)
  - Memory reference:
    - LMD ←Mem[ALUoutput] or
    - Mem[ALUoutput] ←B
  - Branch:
    - If (cond) PC ←ALUoutput
- WB (Write back cycle)
  - Register-Register ALU instruction
    - Regs[rd] ← ALUoutput;
  - Register-Immediate ALU instruction
    - Regs[rt] ←ALUoutput;
  - Load Instruction:
    - Regs[rt] ←LMD;

# Multi-cycle Instruction Execution



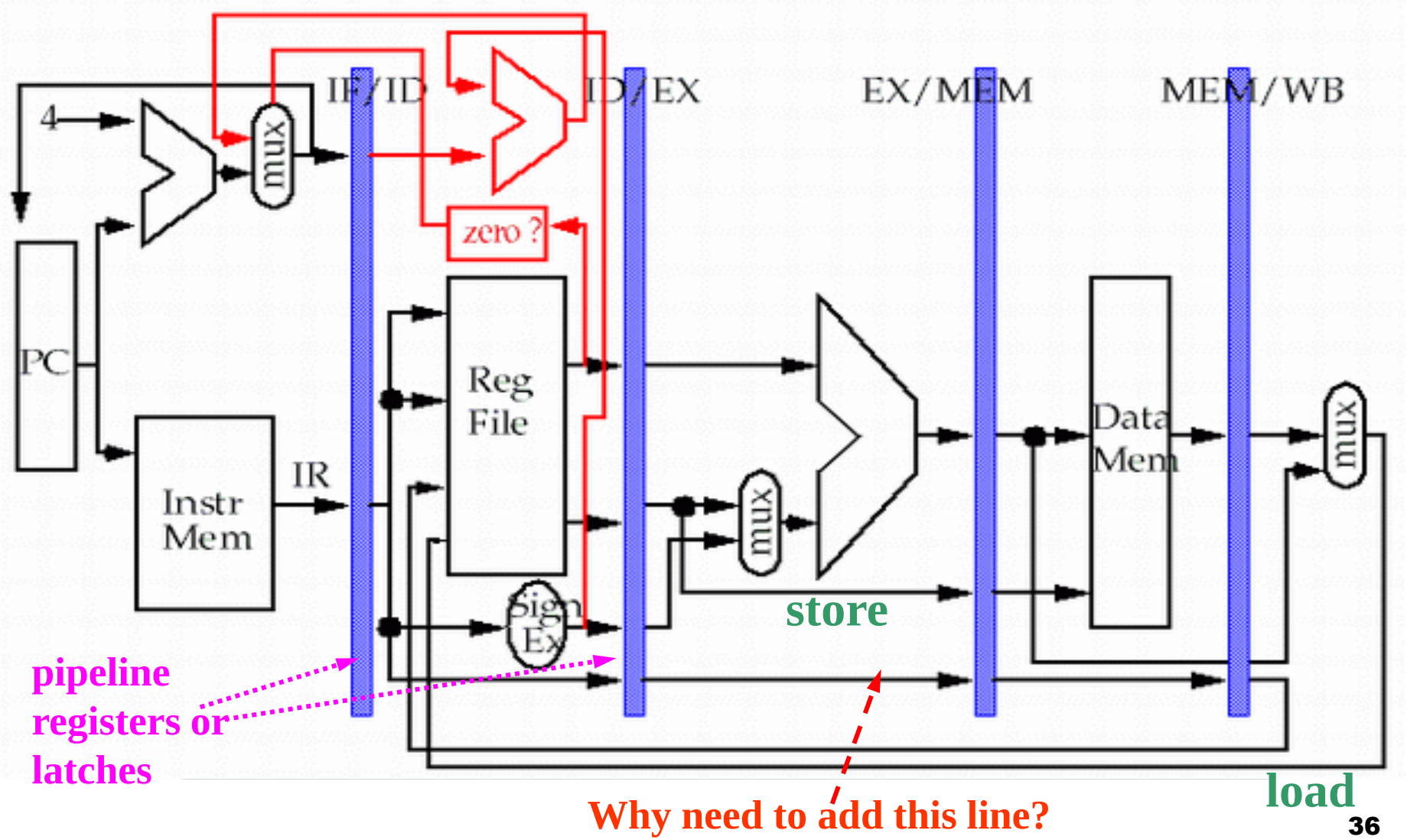
# 3.2.3 The MIPS pipelining



# Table: Events on every stage

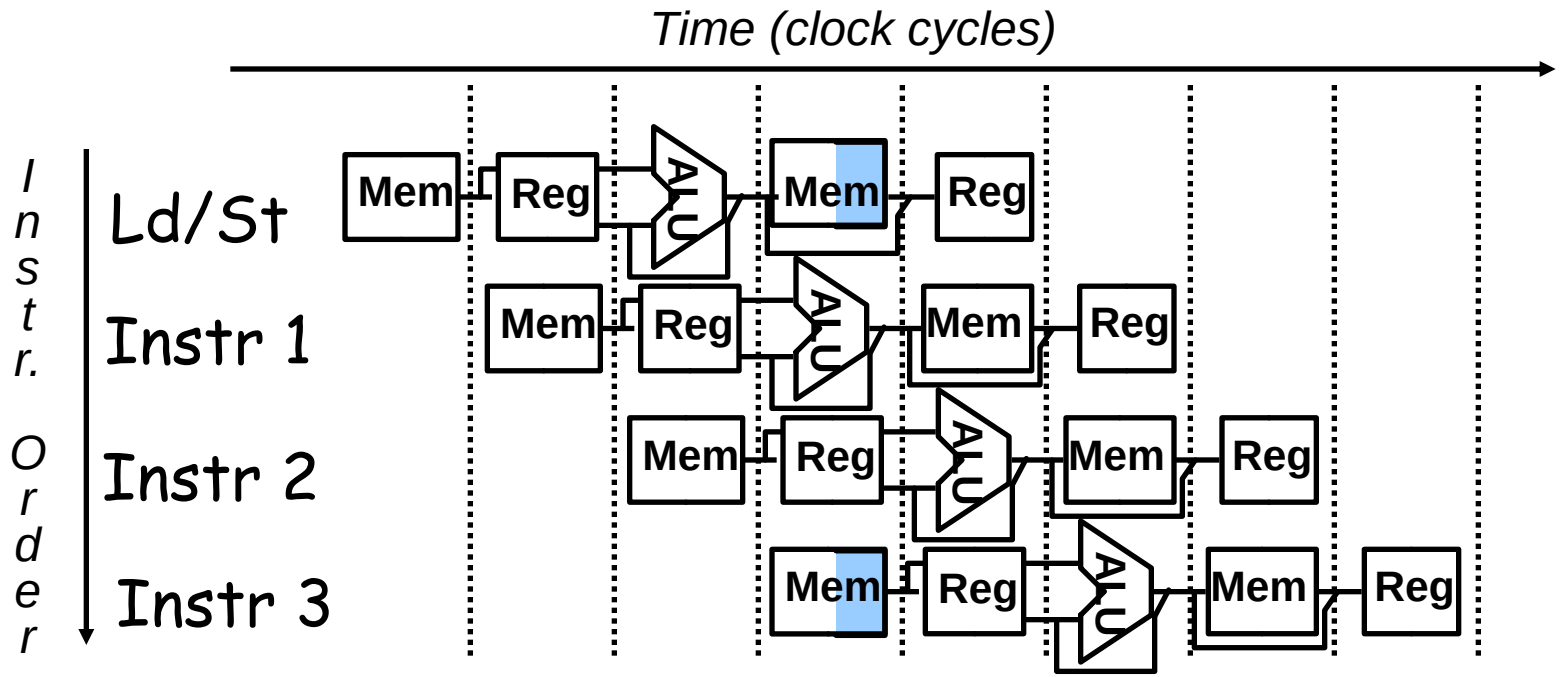
| Stage | Any instruction | | |
|---|---|---|---|
| IF | IF/ID.IR←Mem[PC];<br>IF/ID.NPC, PC ←(if ((EX/MEM.opcode==branch)&EX/MEM.cond)<br>{ EX/MEM.ALUoutput} else {PC+4}); | | |
| ID | ID/EX.A ←Regs[IF/ID.IR[rs]]; ID/EX.B ←Regs[IF/ID.IR[rt]];<br>ID/EX.NPC ←IF/ID.NPC; ID/EX.IR ←IF/ID.IR;<br>ID/EX.Imm ←sign-extend(IF/ID.IR[immediate field]); | | |
| | ALU instruction | Ld/st instruction | Branch instruction |
| EX | EX/MEM.IR ←ID/EX.IR;<br>EX/MEM.ALUoutput ←ID/EX.A func ID/EX.B;<br>or<br>EX/MEM.ALUoutput ←ID/EX.A op ID/EX.Imm; | EX/MEM.IR ←ID/EX.IR;<br>EX/MEM.ALUoutput ←ID/EX.A + ID/EX.Imm;<br>EX/MEM.B ←ID/EX.B; | EX/MEM.ALUoutput ←ID/EX.NPC + (ID/EX.Imm<<2);<br>EX/MEM.cond ←(ID/EX.A==0); |
| MEM | MEM/WB.IR ← EX/MEM.IR;<br>MEM/WB.ALUoutput ←EX/MEM.ALUoutput; | MEM/WB.IR ← EX/MEM.IR;<br>MEM/WB.LMD ←Mem[EX/MEM.ALUoutput];<br>Or MEM/WB.LMD ←Mem[EX/MEM.ALUoutput]; | |
| WB | Regs[MEM/WB.IR[rd]] ←MEM/WB.ALUoutput; or<br>Regs[MEM/WB.IR[rt]] ←MEM/WB.ALUoutput; | For Load only;<br>Regs[MEM/WB.IR[rt]] ←MEM/WB.LMD | |

# Advanced pipeline



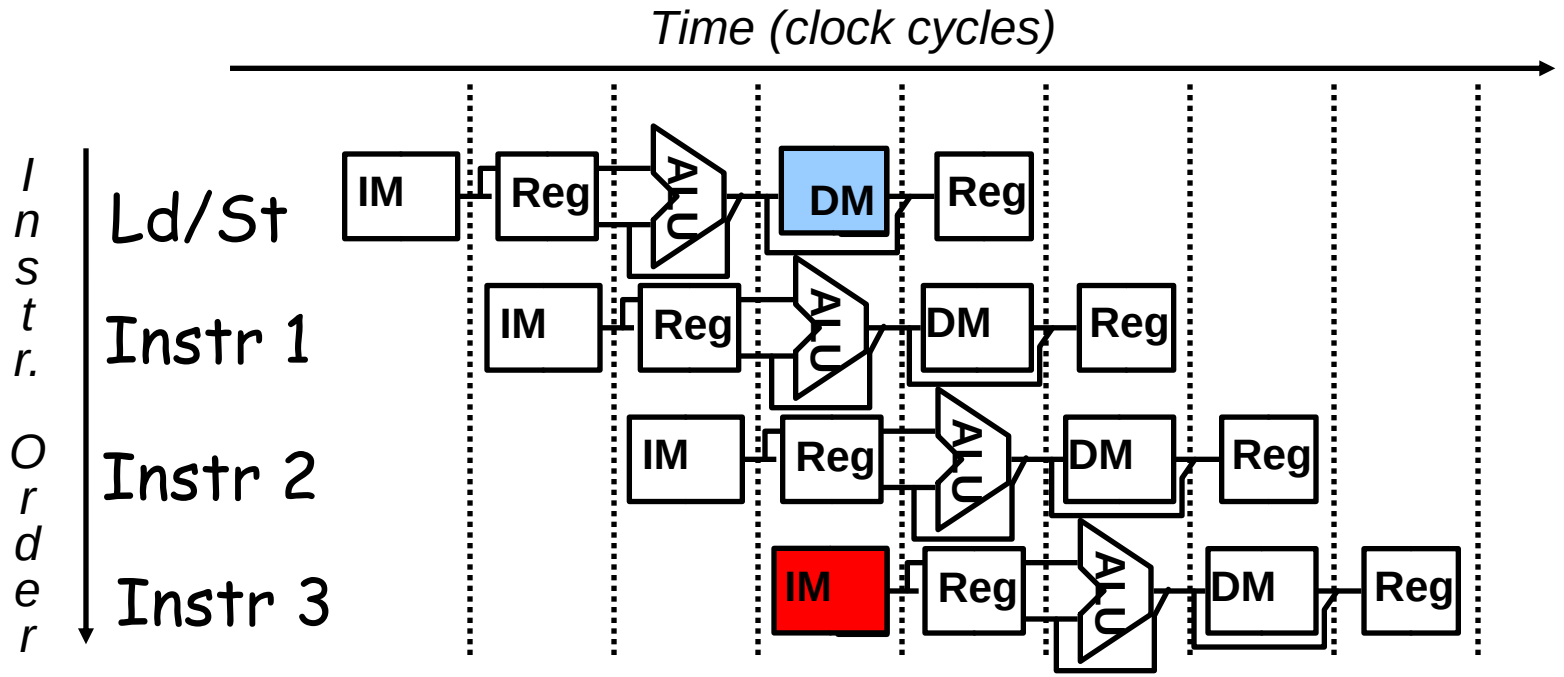
# Problems that pipelining introduces

一、There is conflict about the memory !



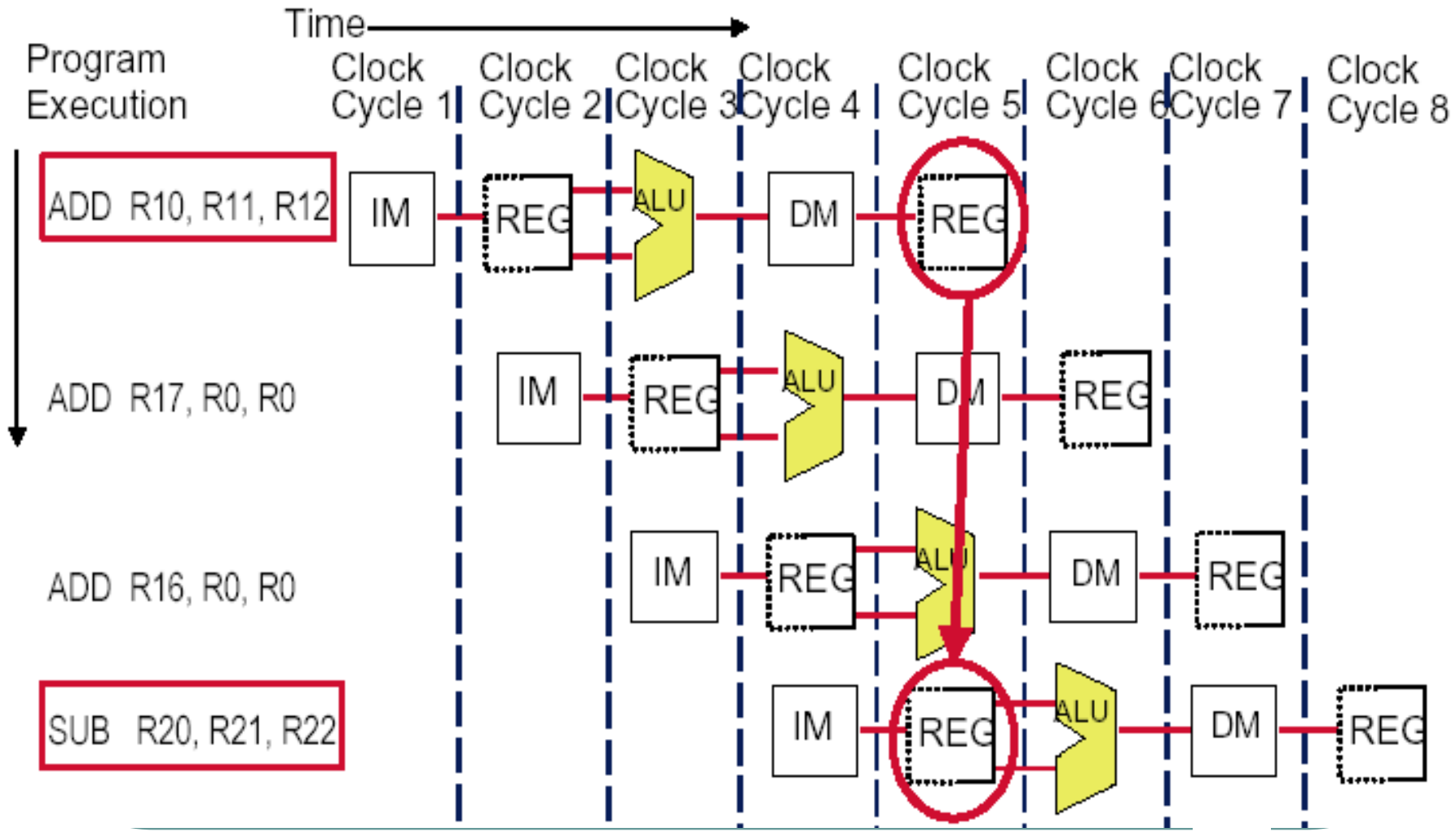
# Separate instruction and data memories

- use split instruction and data cache



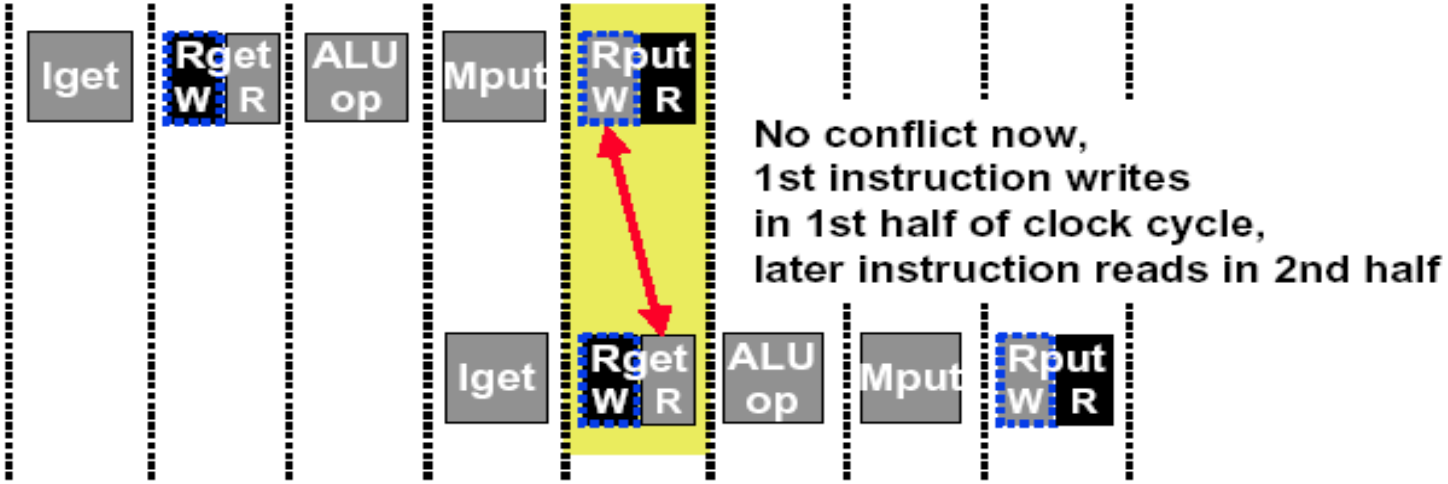

- the memory system must deliver 5 times the bandwidth over the unpipelined version.

# 二、The conflict about the registers !



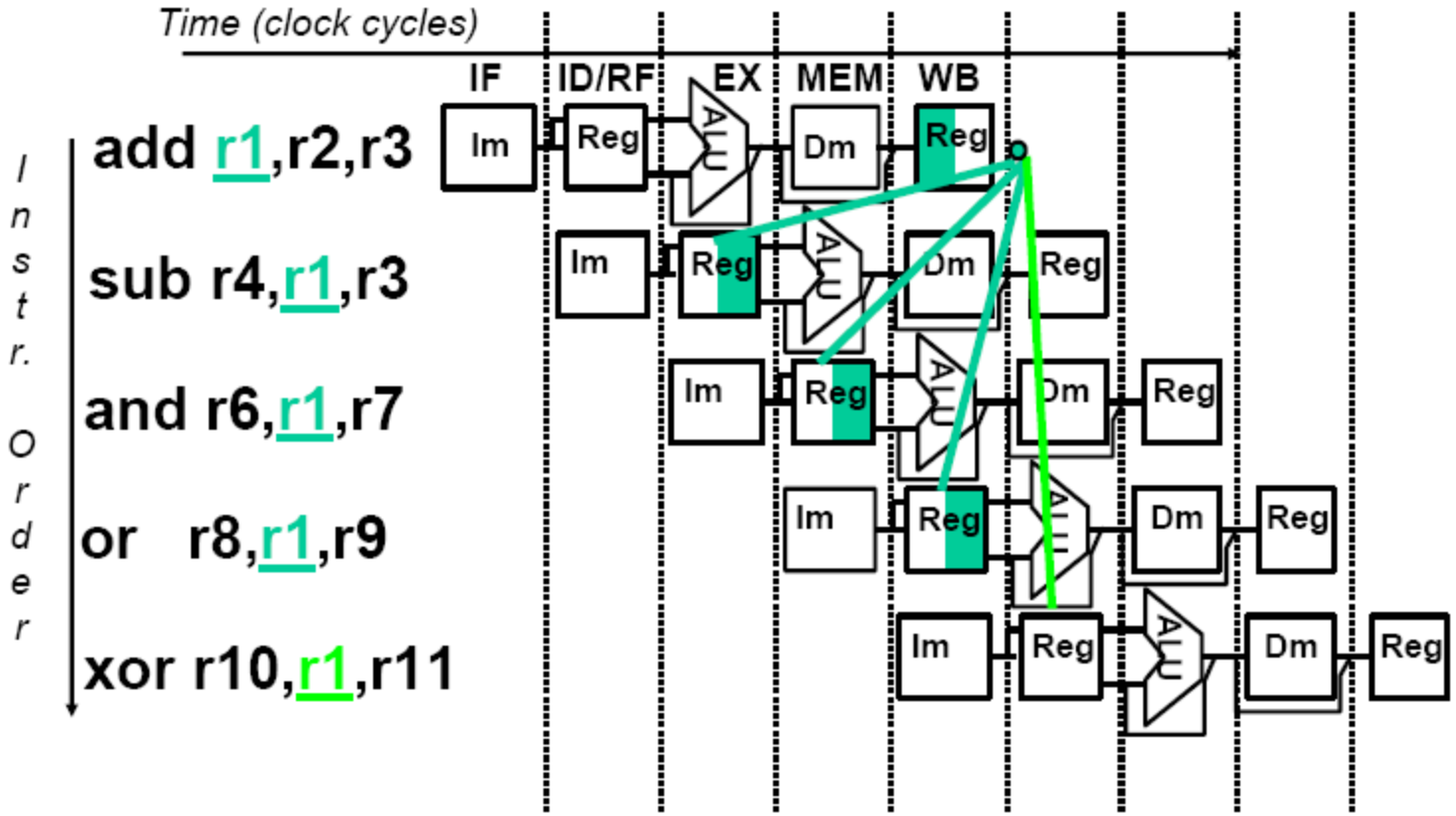
# Sometimes we can redesign the resource

- Allow WRITE-then-READ in one clock cycle (double pump)

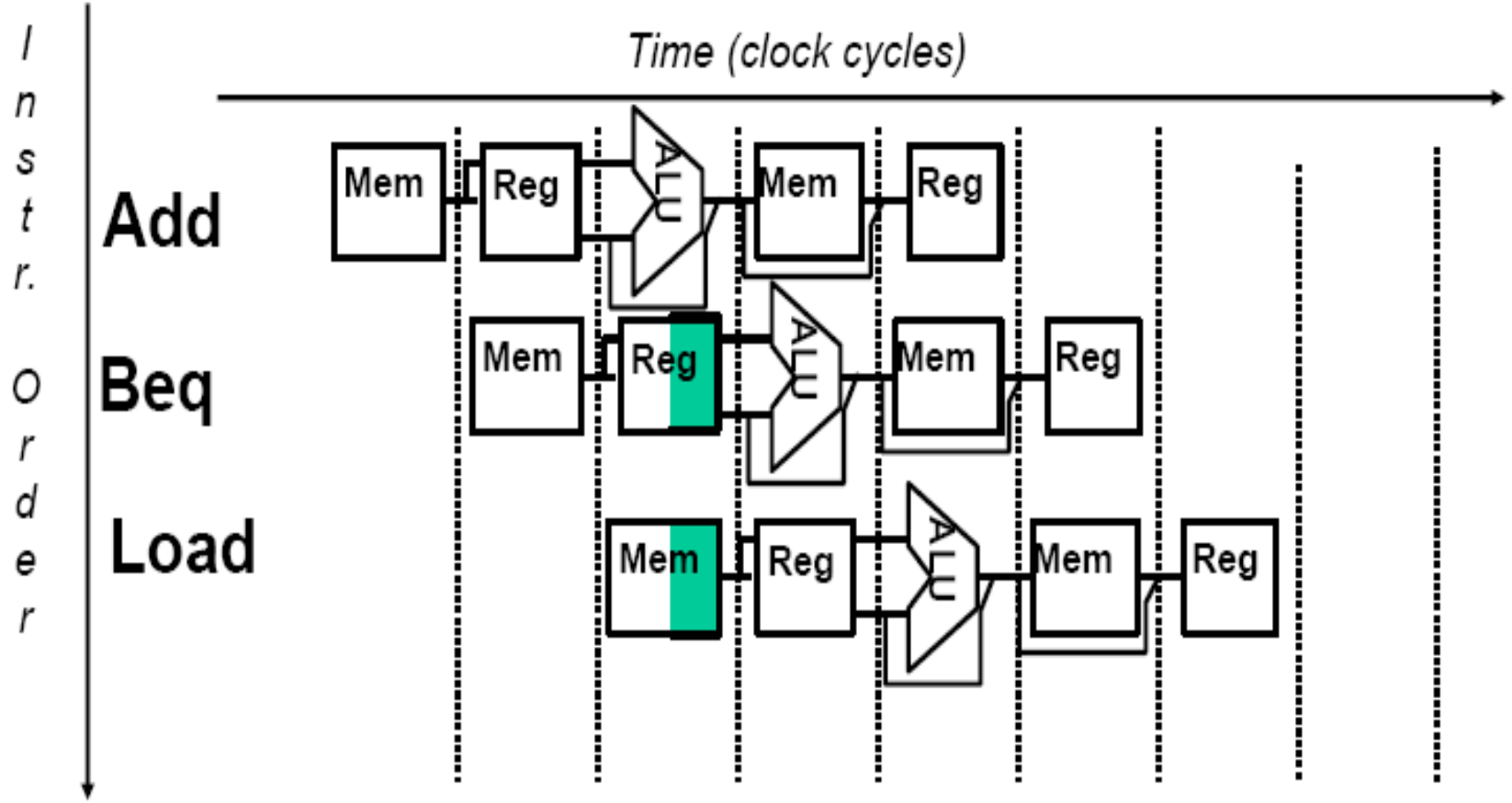



  - Two reads and one write required per clock.
  - Need to provide two read port and one write port.

# 三、The conflict about the datas !



# 四、Conflict occurs when PC update



# 五、Must latches be engaged ? Yeah !

- Ensure the instructions in different stages do not interfere with one another .
- Through the latches, can the stages be combined one by one to form a pipeline.
- The latches are the pipeline registers , which are much more than those in multi-cycle version
  - IR: IF/ID.IR; ID/EX.IR; EX/DM.IR; DM/WB.IR
  - B: ID/EX.B; EX/DM.B
  - ALUoutput: EX/DM.ALUoutput, DM/WB.ALUoutput

# 3.3 The Major Hurdle of Pipelining—Pipeline Hazards

*本科回顾------- Appendix A.2*

3.3.1 Taxonomy of hazard

3.3.2 Performance of pipeline with Hazard

3.3.3 Structural hazard

3.3.4 Data Hazards

3.3.5 Control Hazards