6 Pipelined Processor

6.1 The Principle of Pipelined Processor

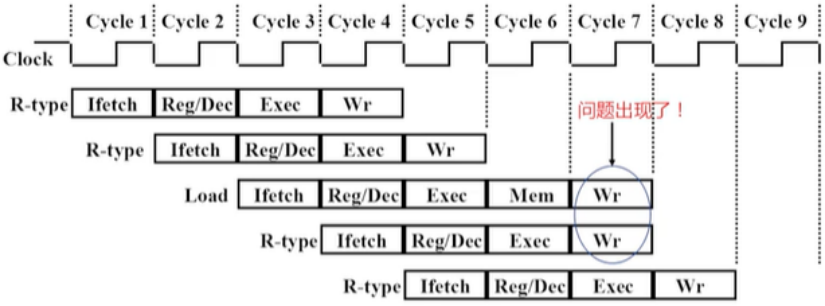
5 Pipeline stage

- **Instruction Fetch**: fetch instruction, increase `PC` by 4;
- **Register Access & Instruction Decode**: access register and perform instruction decode;
- **Instruction Execution**: execute instruction;
- **Memory Access**: read/write memory;
- **Write Back**: Write the result to registers.

Pipelined MIPS

- Instructions have the same length: fetch instructions at stage 1 and decode at stage 2;
- Simple instruction format (3 types): access registers at stage 2;
- Only `load` and `store` instructions can access memory: calculate memory address at stage 3;
- Each instruction writes at most one result: perform writing operations at stage 4 (MEM) or stage 5 (WB).

Resources Conflict: Pipelined R-Type instructions only need 4 stages (do not need to access memory), but `load` instruction needs 5 stages. If we simply execute them in pipeline, then will cause resources conflict.

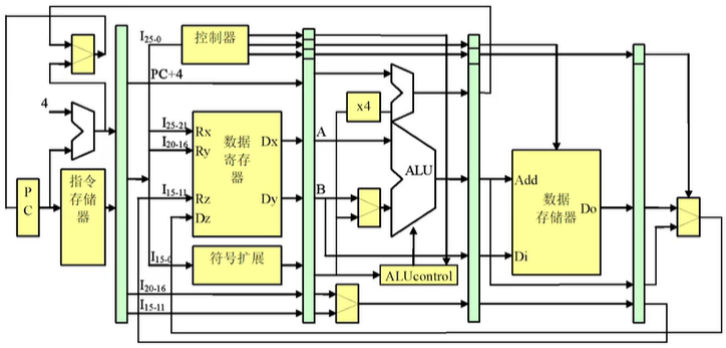


Two instructions want to access the write pin of registers.

Solution to Resources Conflict: Every instructions have the same number of stages .

- Let the memory access stage of R-Type instructions be empty operations;
- Let the write back stage of `store` instructions be empty operations;
- Let the write back stage of `beq` instructions be empty operations, and write the result to `PC` in the memory access stage.

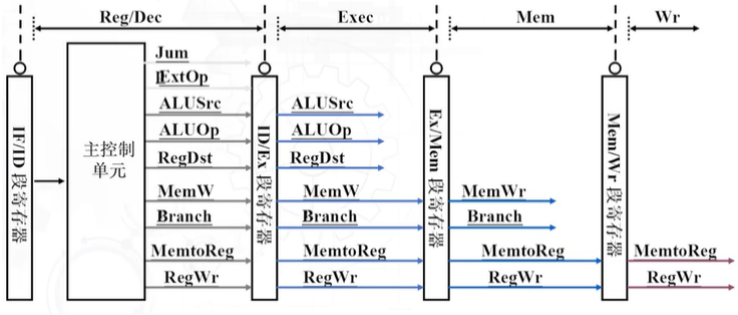
6.2 The Implementation of Pipelined Processor



There are **segment registers**（段寄存器） in the picture above (the green line).

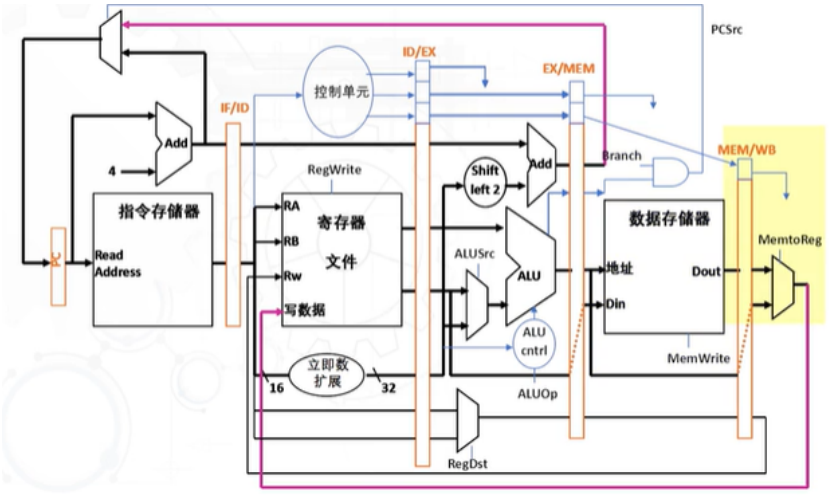
Control Signals

- Use the main control unit to generate all *signals* in ID (Instruction Decode) stage, and the signals will be passed by segment registers.



- Signals that every stage needs:
  - **Instruction Fetch**: no signal is needed.
  - **Instruction Decode**
    - `ExtOp`: perform zero-extension or sign-extension;
    - `Jump`: ready to perform modification to `PC` if the instruction is jump instruction.
  - **Execution**
    - `ALUSrc`: ALU needs to know whether its input is from the bit-extender or from bus B;
    - `ALUop`: ALU needs to know which operation to perform;
    - `RegDst`: Choose the correct register address. Use this signal in this stage can reduce the length of segment registers.
  - **Memory Access**
    - `MemWr`: memory unit needs to know whether the instruction will write to the memory;
    - `Branch`: the destination address is calculated by ALU in execution stage, therefore the `Branch` signal will be used after the execution stage completes.
      - Its destination address will be in the input pin of PC.
      - When the next instructions executed in Instruction Fetch stage, the value of PC will be updated and the correct instruction will be read according to the new PC.
  - **Write Back**
    - `MemtoReg`: The register unit needs to know its input is from memory or ALU.
    - `RegWr`: the register unit needs to know whether the instruction will write back to registers.

The Full Implementation of Pipelined Processor

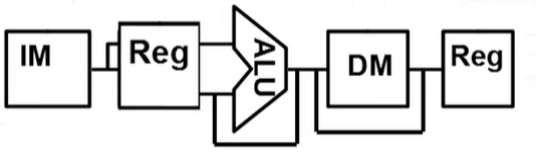


Bits of Segment Registers

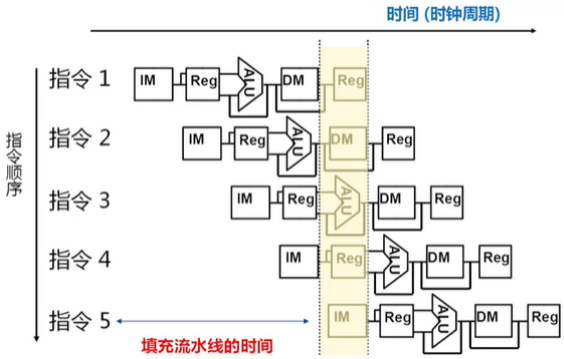
- PC: $32\ \text{bits}$;
- IF/ID: $32\ \text{bits} \times 2 = 64\ \text{bits}$; (Instruction and the address of next instruction)
- ID/EX: $9\ \text{bits} + 32\ \text{bits} \times 4 + 5\ \text{bits} \times 2 = 147\ \text{bits}$ (Signals, the address of next instruction, the immediate number, the value of `rs`, the value of `rt`, the number of `rt` and the number of `rd`)
- EX/MEM: $4\ \text{bits} + 1\ \text{bits} + 32\ \text{bits} \times 3 + 5\ \text{bits} = 106\ \text{bits}$ (Signals, whether the result is zero, the output of ALU, the address of next instruction if the instruction is a branch instruction, the number of `rt` and the destination register address).
- MEM/WB: $2\ \text{bits} + 32\ \text{bits} \times 2 + 5\ \text{bits} = 71\ \text{bits}$ (Signals, the output of ALU, the output of memory, the destination register address).

6.3 Relations and Hazards

The simple notations of five-stage pipelined processor.



The pipelined processor can improve its throughput（吞吐率）. An extra time in the beginning is needed to fill the pipeline.



Once the pipeline is full, one instruction will finish executing every cycle, this time we have the best CPI (cycle per instruction) of 1.

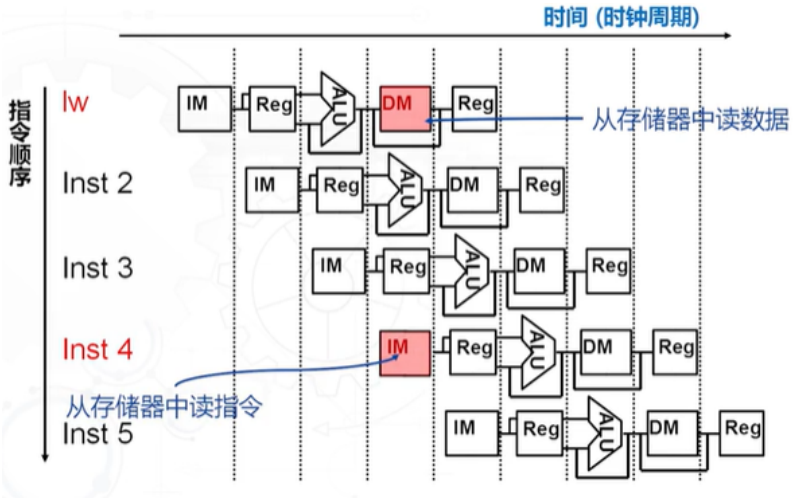
Relation: some instructions have some relations（相关性） with the others.

Hazards

- **Structural hazards**: resource-related, the hardware unit is working for the previous instruction.

  General Solution: more hardware resources（增加硬件资源）, and pipelined function units（功能单元流水化）.

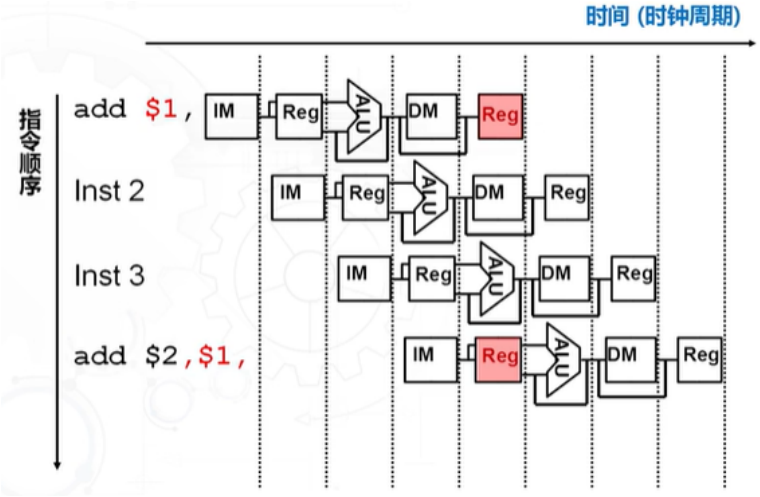
  1. The read and write memory operations may be overlapped.



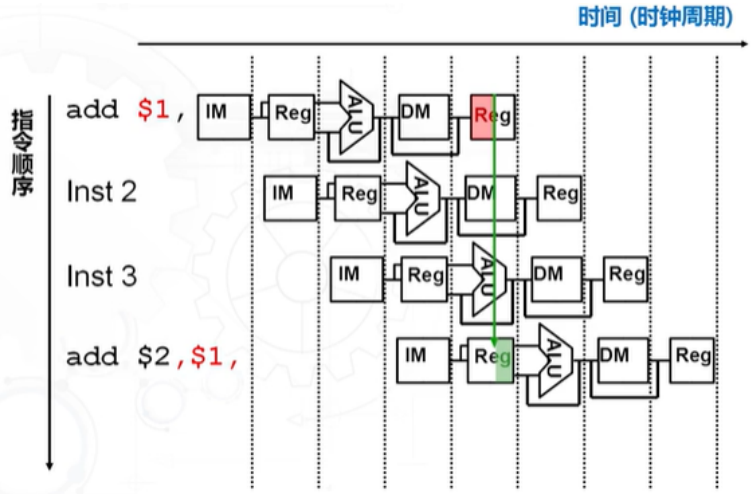
     Solution:

     - Add empty instructions before hazards (in the picture above, add an empty instruction before instruction 4).
     - Separate Instruction memory and Data memory (Nowadays the instruction cache and the data cache are separate in L1 cache).

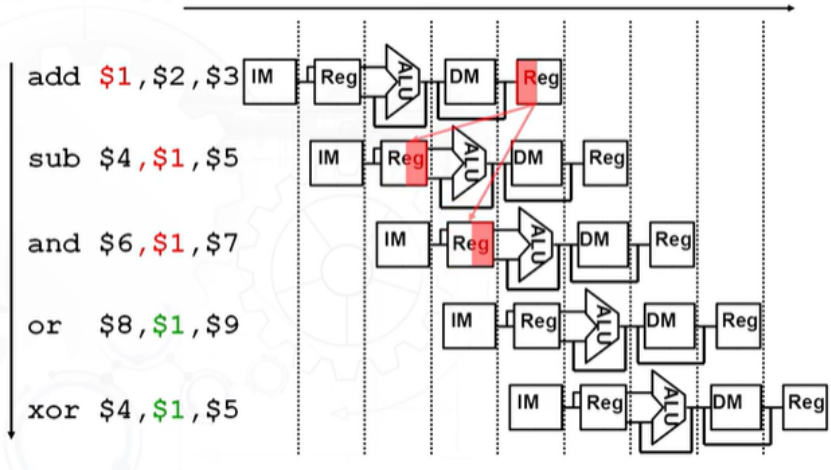
  2. The read and write registers operations may be overlapped.



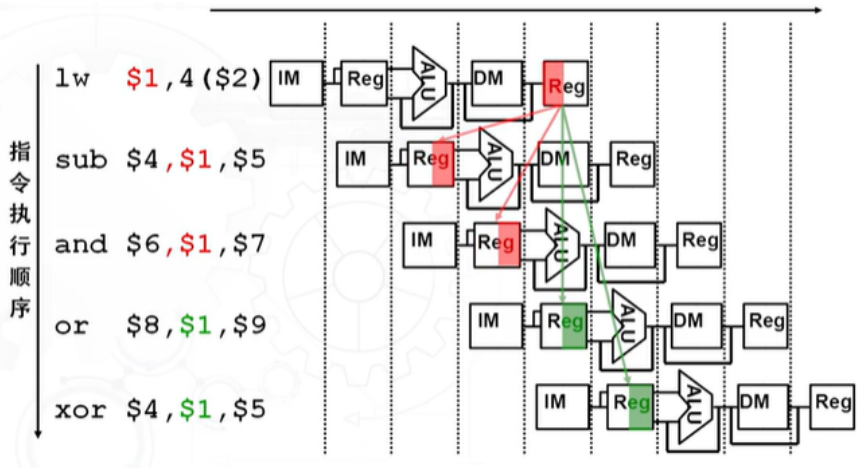
     Solution: This situation can be prevented, since the speed of accessing register is faster than many other operations. The register unit have separate read pins and write pins. We can divide one cycle into two halves. The register-write operations will be performed in the first half, and the register-read operations will be performed in the second half. Therefore, we prevent the structural hazard of register from happening.



- **Data hazards**: data-related, we must wait until the last instruction is finished to get the data we need in this instruction, otherwise the data can be wrong.
    1. **Read after write (register) data hazard**（（寄存器）写后读数据冒险）

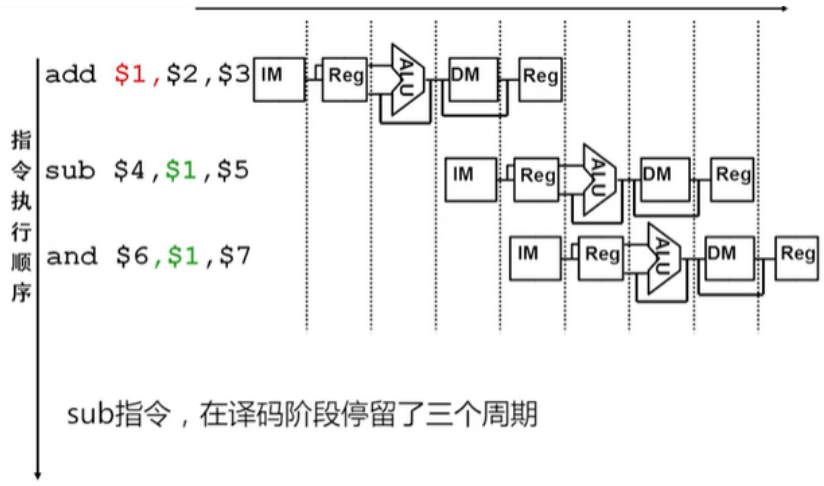


    2. **Load-use data hazard**（读存储器-使用数据冒险）

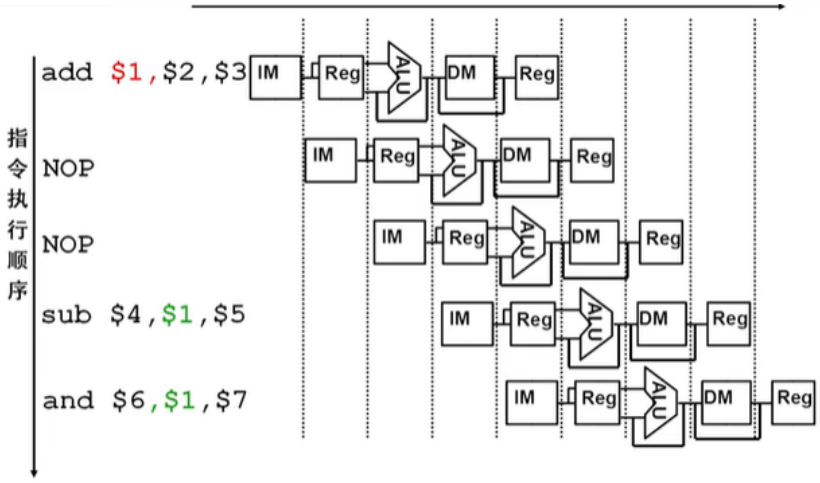


    General Solution:

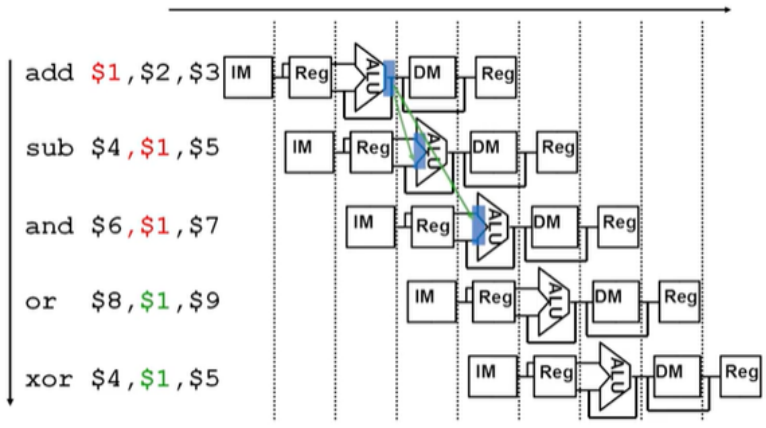
    - **Pipeline pause**（流水线停顿）: Need to add a detector in circuit (hardware) to detect whether the current instruction conflicts with the previous instructions. If so, then pause the current instruction until there is no more conflicts.



- **Add empty instructions (compiler)**: The compiler can also add empty instructions to avoid data hazard while compiling the high-level language to assembly codes.

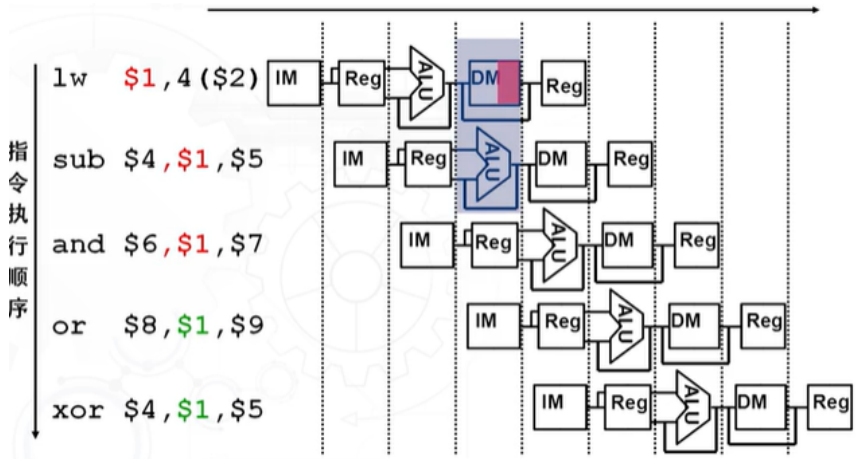


- **Forwarding**（前向传递）: Transfer the current data directly to the next few instructions after ALU completes the calculations, without waiting until the WB stage. This special data path is called *forward data path*（前向数据通路）. Send the result of stage EX to beginning of EX of the next two instructions, and use logic selector to choose the correct data.



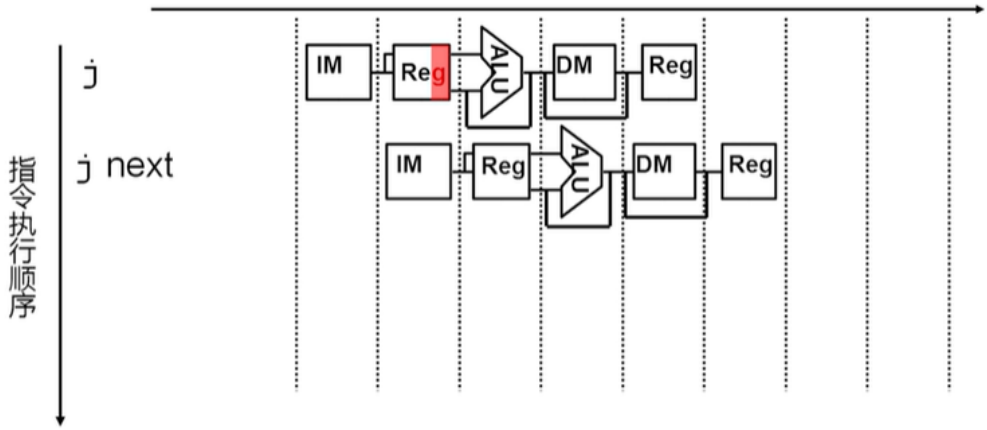
But forwarding cannot solve all the data hazard problems, for example, it cannot solve the data hazard between the instruction `lw` and `sub`.

Actually, forwarding cannot solve some of the load-use data hazard problems because memory access operations are at stage 4 (MEM) and the execution operations are at stage 3 (EX). Therefore, sometime we need to add an empty instruction to avoid this situation.

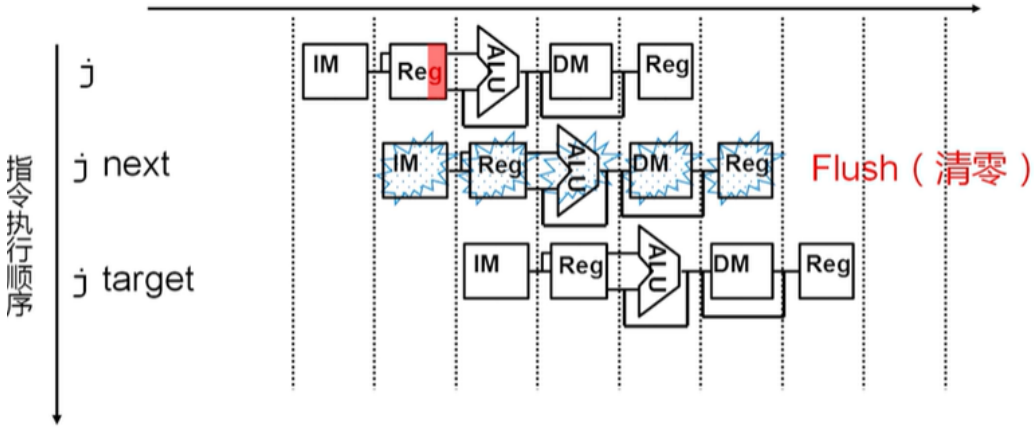


- **Control hazards**: mainly caused by *jump/branch* instructions, the address of the next instruction will be determined after the last instruction is finished, otherwise the address can be wrong.

    1. **Unconditional jump instruction control hazard**

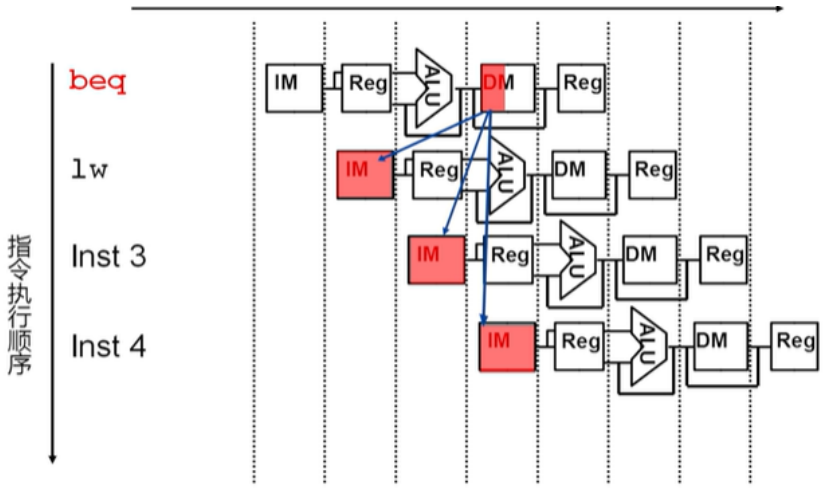


Solution: if we detect this kind of control hazards, then just flush（清零） the next instruction and jump to the right address and perform the correct next instruction.



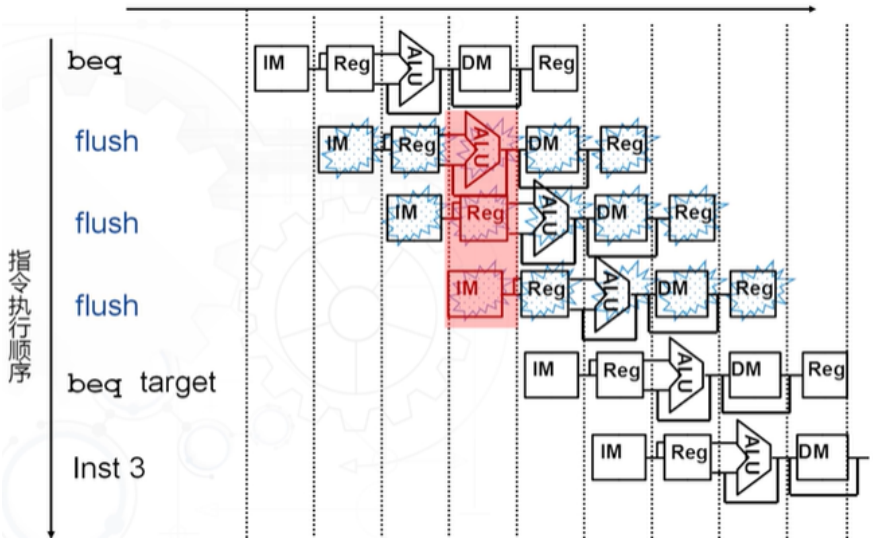
Cost: one more cycle to use, $CPI_{jump} = 2$.

2. **Conditional jump instruction control hazard**: This kind of control hazard will affect CPI seriously.



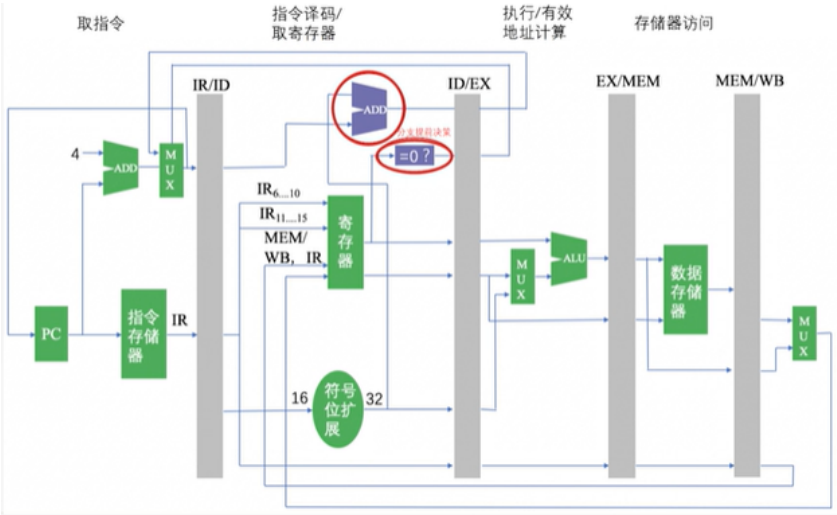
Solution

- Flush the wrong instructions（清零）.

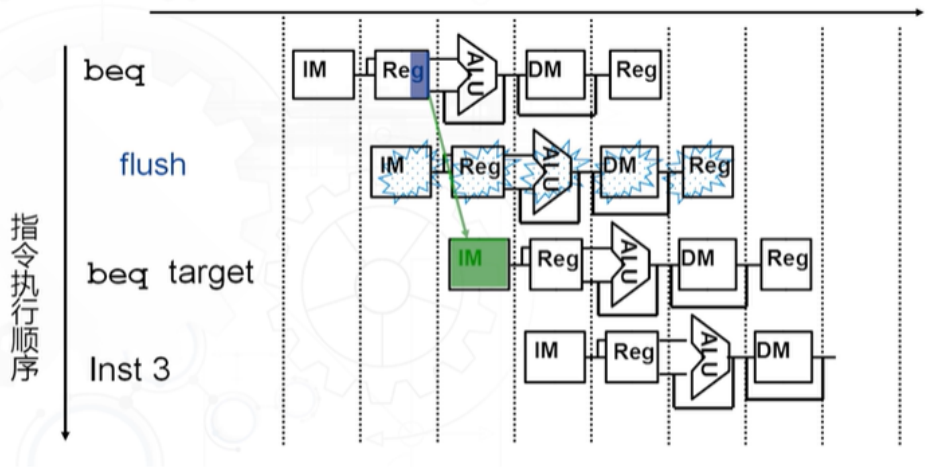


Cost: $CPI_{beq} = 4$. Too much.

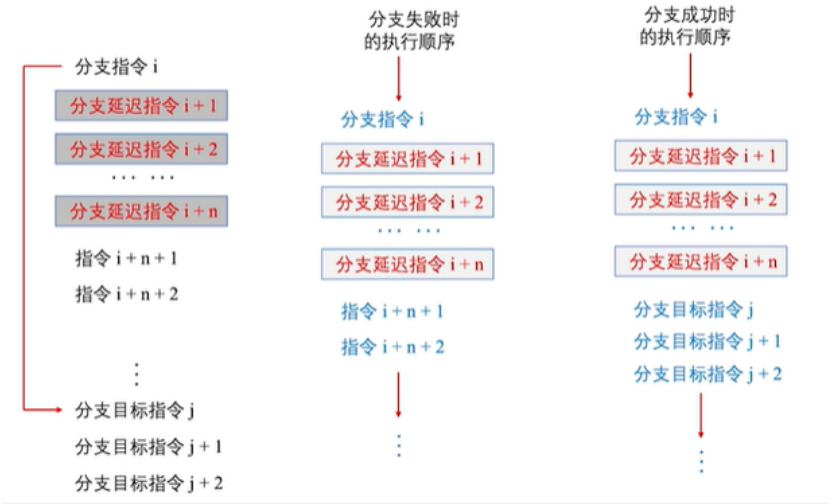
- **Earlier Selection**（分支提前决策）: add a selector and an adder at ID stage and choose the correct branch as soon as possible. This will reduce the failure cost.



The result is as follows. We only have to flush the next instruction. Therefore, we reduce the CPI by 2.



- **Branch Slot (compiler)**（转移延迟槽）: Put one or more instructions that are bound be executed after the branch instruction, which can hide the latency of control hazard. This optimization needs compiler support.



The size of the branch slot n is usually the latency of a branch instruction. For example, in five-stage normal pipeline, $n = 3$; in five-stage earlier-selection pipeline, $n = 1$.

The compiler will be more complex.

6.4 The Optimization of Pipeline

Five-stage Instruction Pipeline

- Ideal case: the pipeline does not pause, then $CPI = 1$.
- **Pros**: Simple and easy to implement;
- **Cons**: the limit of performance is $CPI = 1$; difficult to optimize complex instructions like multiplications.
- **Tightly Coupled**（紧耦合）: the pause of one instruction will cause the following few instructions to pause.

Performance

$$CPI = CPI_{ideal} + CPI_{stall}$$

The reasons of CPI_{stall}:

- Data hazard;
- Structure hazard;
- Control hazard;

- Memory access latency.

Improvement of Processor

- **Super-pipelining**（超级流水）: reduce the time period of a cycle (increase the frequency of the CPU clock).

  The depth of pipeline should not be too large.

  > [*Example*] Suppose the segment registers' latency is $50\ \text{ps}$, and latency of a stage is $200\ \text{ps}$. Then the latency of an instruction is $(200\ \text{ps} + 50\ \text{ps}) \times 5 = 1250\ \text{ps}$.
  >
  > If we use 10-stage pipeline, then the latency of a stage will be $100\ \text{ps}$. The latency of an instruction is $(100\ \text{ps} + 50\ \text{ps}) \times 10 = 1500\ \text{ps}$.

  Other Problems

  - Increase the complexity and cost;
  - Bad performance:
    - The latency of segment registers;
    - More hazards and pause.
  - High power-consuming（功耗增加）.

- **Multiple-issue**（多发射）: fetch and execute some instructions parallelly. It can use the instruction level parallelism (ILP) to reduce CPI and make $CPI < 1$.
  - **Static Multiple-issue**（静态多发射）: The compiler chooses the instructions that can be executed parallelly.
  - **Dynamic Multiple-issue**（动态多发射）: Superscalar processors（超标量处理器）. Choose the instructions that can be executed parallelly dynamically (not by compiler, by hardware).
    - In-order（按序）Execution.
    - Out-of-order（乱序）Execution.
  - Will be discussed further in Chapter 8.