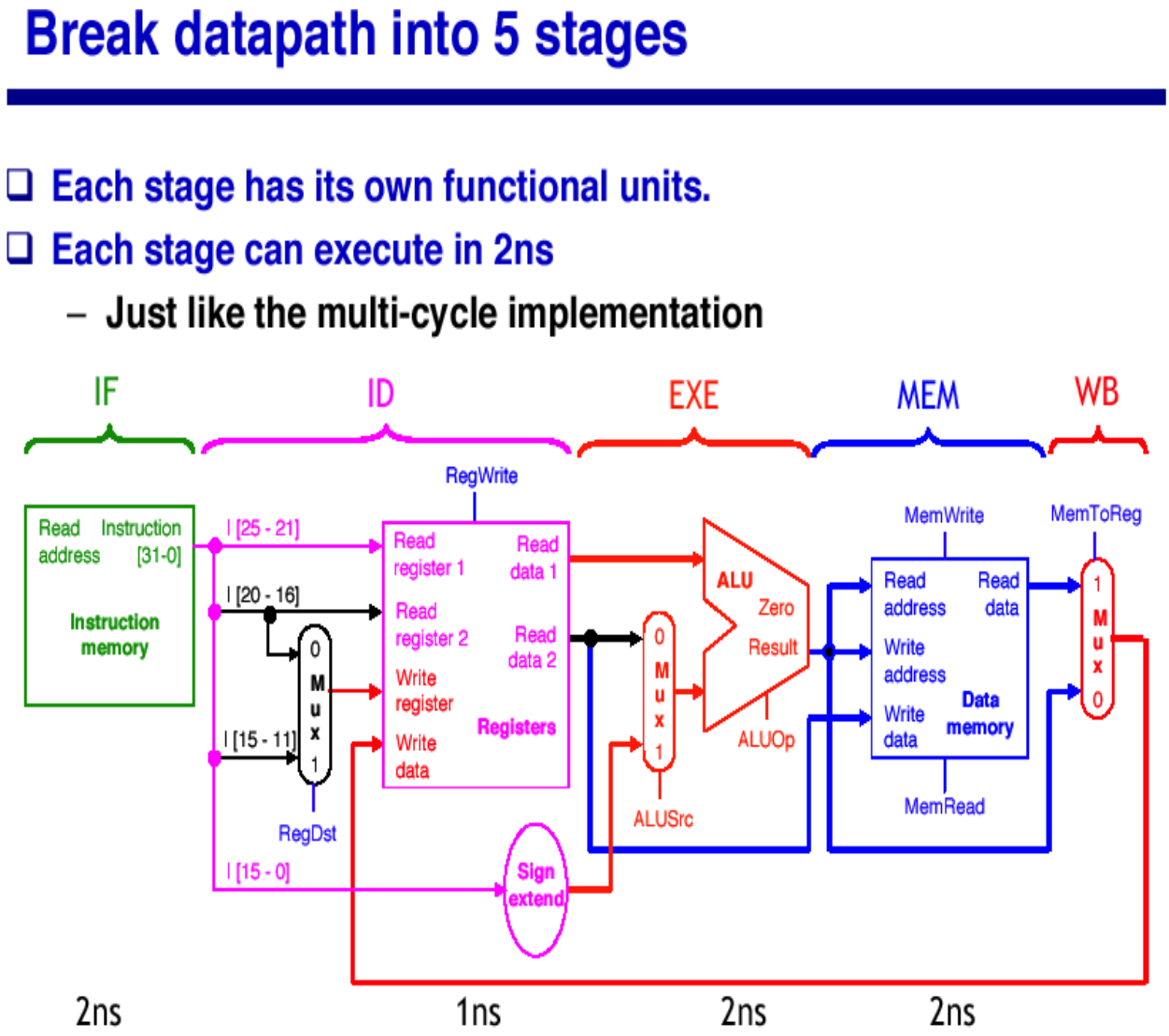
# Pipelined MIPS

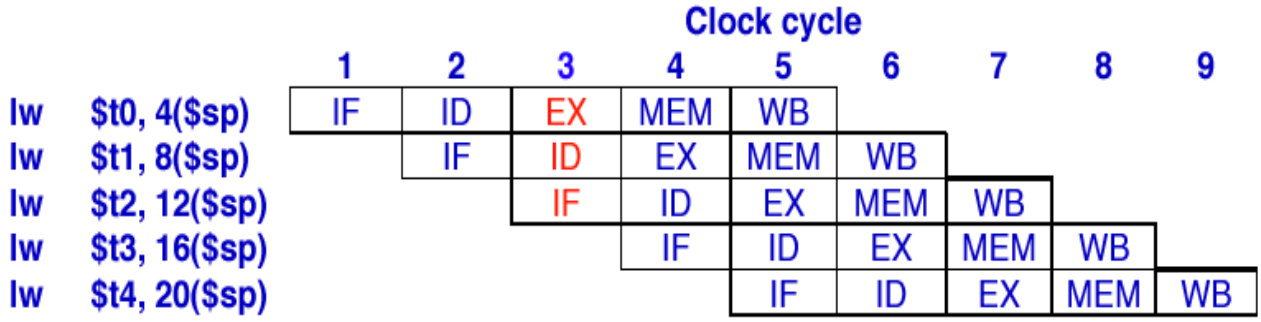
While a typical instruction takes 3-5 cycles (i.e. 3-5 CPI), a pipelined processor targets 1 CPI (at least gets close to it).




It shows the rough division of responsibilities.

The buffers between stages are not shown.

# Pipelining Loads




- **A pipeline diagram shows the execution of a series of instructions.**
  - The instruction sequence is shown vertically, from top to bottom.
  - Clock cycles are shown horizontally, from left to right.
  - Each instruction is divided into its component stages. (We show five stages for every instruction, which will make the control unit easier.)

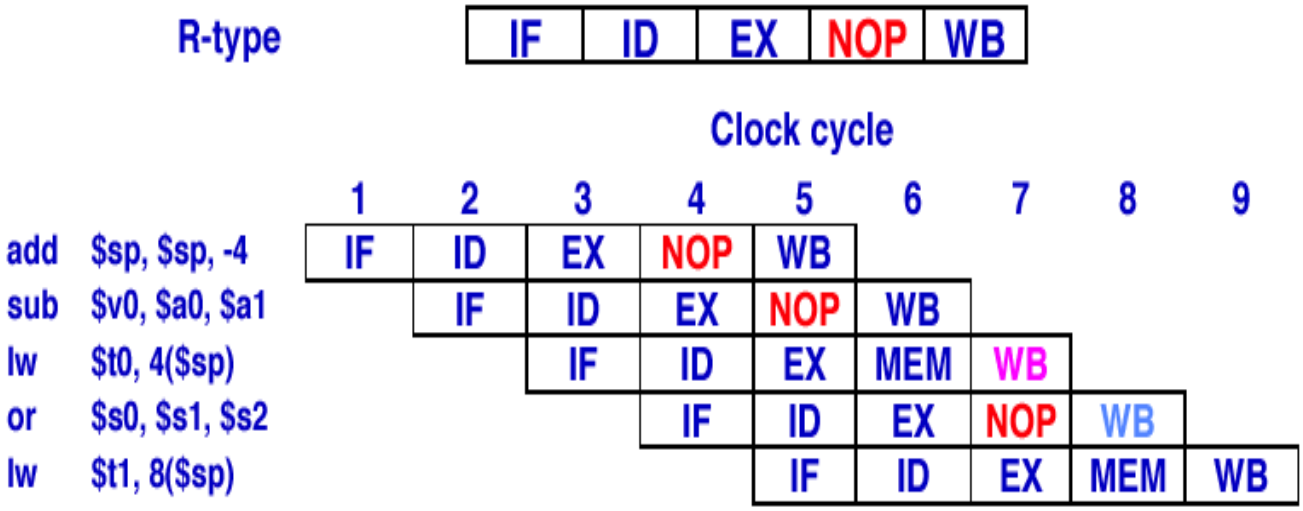
- **This clearly indicates the overlapping of instructions. For example, there are three instructions active in the third cycle above.**
  - The "lw $t0" instruction is in its Execute stage.
  - Simultaneously, the "lw $t1" is in its Instruction Decode stage.
  - Also, the "lw $t2" instruction is just being fetched.

Problem 1. How can the same adder perform IF and EX in cycle 3? We need an extra adder! Gradually we need to modify the data path for the multi-cycle implementation.

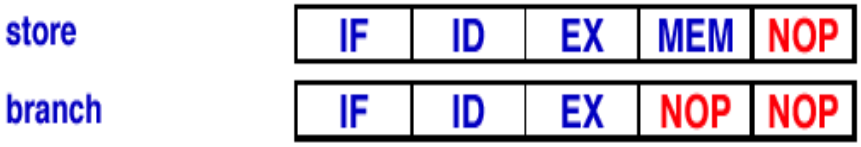
Problem 2. How can we read instruction memory and data memory in the same clock cycle? We had to return to Harvard architecture!

# Uniformity is simplicity

- **Enforce uniformity**
  - Make all instructions take 5 cycles.
  - Make them have the same stages, in the same order
    - Some stages will do nothing for some instructions




- Stores and Branches have NOP stages, too…



# Speedup

The steady state throughput is determined by the time t needed by one stage.

The length of the pipeline determines the pipeline filling time

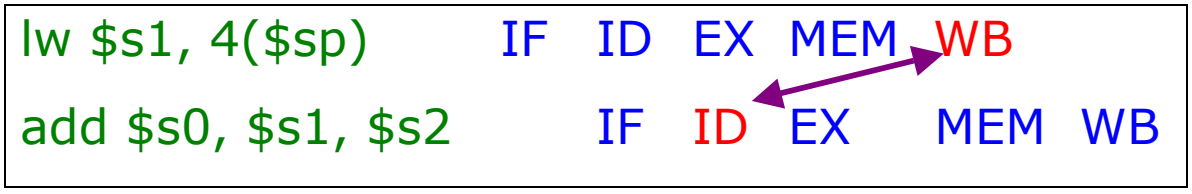
If there are k stages, and each stage takes t time units, then the time needed to execute N instructions is

$$k.t + (N-1).t$$

*Estimate the speedup when N=5000 and k=5*

# Hazards in a pipeline

Hazards refer to conflicts in the execution of a pipeline. On example is the need for the same resource (like the same adder) in two concurrent actions. This is called structural hazard. To avoid it, we have to replicate resources. Here is another example:




Notice the second instruction tries to read $s1 before the first instruction complete the load! This is known as data hazard.

# Avoiding data hazards

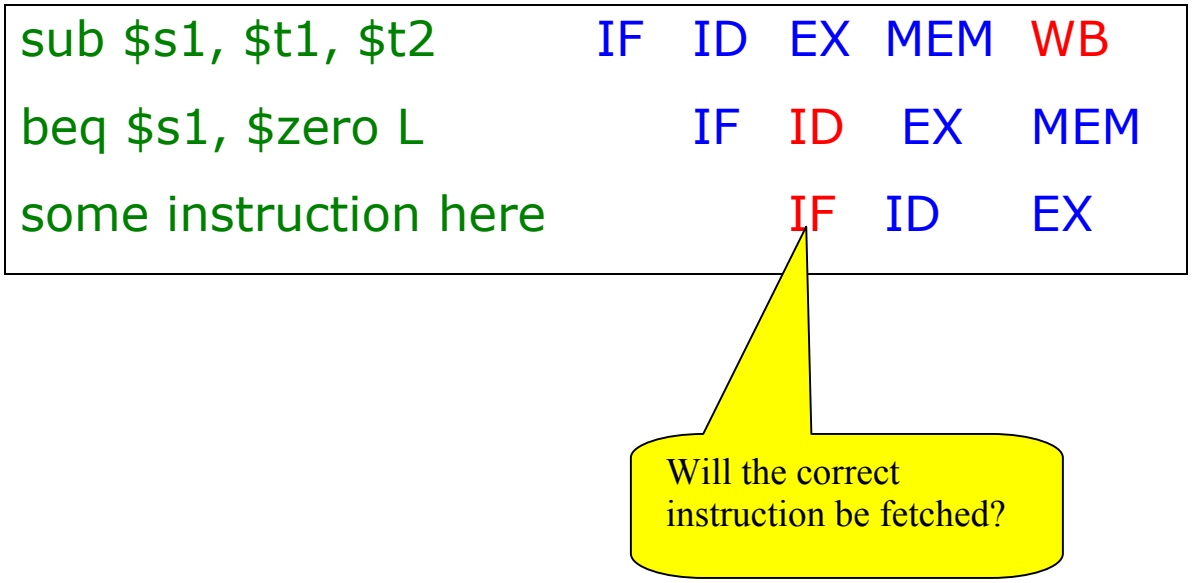
One solution is in insert bubbles (means delaying certain operation in the pipeline)

| | | | | | | |
|---|---|---|---|---|---|---|
| lw $s1, 4($sp) | IF | ID | EX | MEM | WB | |
| add $s0, $s1, $s2 | | IF | nop | nop | nop | ID |

Another solution may require some modification in the datapath, which will raise the hardware cost

Hazards slow down the instruction execution speed.

# Control hazard




There is no guarantee! The next instruction has to wait until the predicate ($s1=0) is resolved. Look at the tasks performed in the five steps – the predicate is evaluated in the EX step. Until then, the control unit will insert **nop (also called bubbles)** in the pipeline.

## The Five Cycles of MIPS

(Instruction Fetch)

IR:= Memory[PC]; PC:= PC+4

(Instruction decode and Register fetch)

A:= Reg[IR[25:21]], B:=Reg[IR[20:16]]

ALUout := PC + sign-extend(IR[15:0]]

(Execute|Memory address|Branch completion)

Memory refer: ALUout:= A+ IR[15:0]

R-type (ALU): ALUout:= A op B

Branch: if A=B then PC:= ALUout

(Memory access | R-type completion)

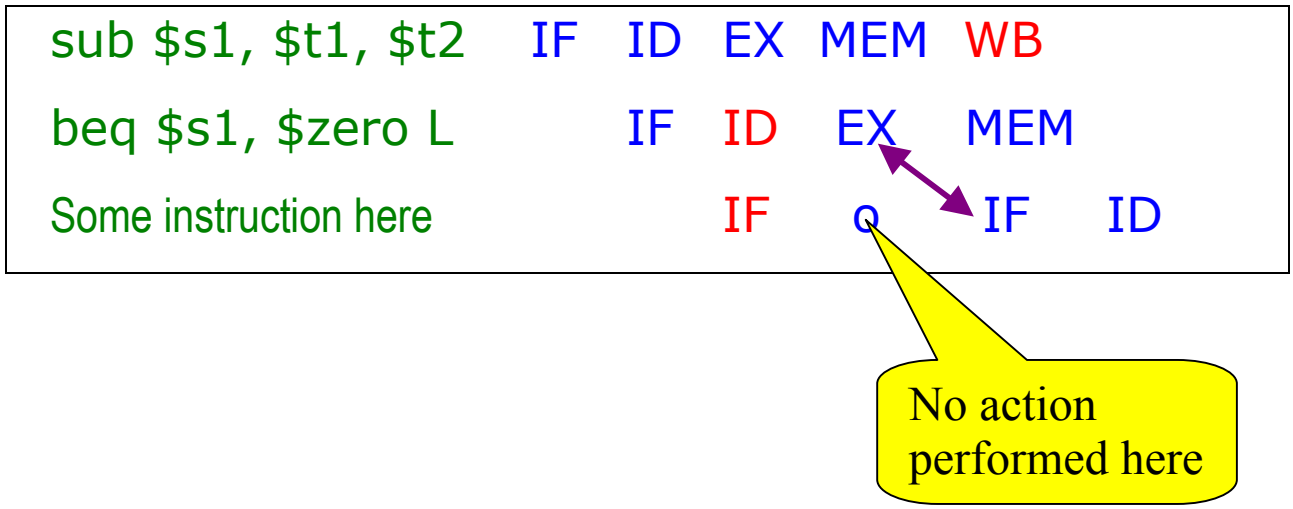
LW: MDR:= Memory[ALUout]

SW: Memory[ALUout]:= B

R-type: Reg[IR[15:11]]:= ALUout

(Write back)

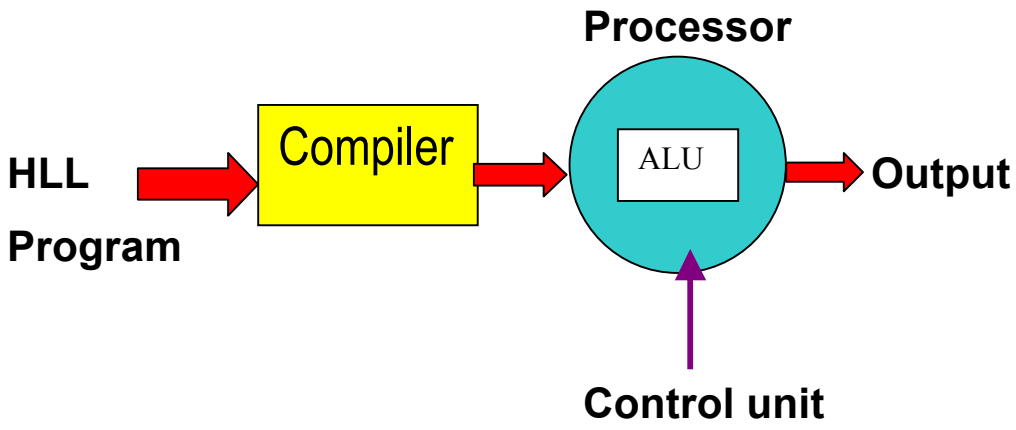
LW: Reg[[20:16]]:= MDR




An alternative approach to deal with this is for the compiler (or the assembler) to insert NOP instructions, or reorder the instructions.

## Dealing with Hazards in Pipelined Processors

## Two options




1. Either the control unit can be smart, i,e. it can delay instruction phases to avoid hazards. Processor cost increases.

2. The compiler can be smart, i.e. produce optimized codes either by inserting NOPs or by rearranging instructions. The cost of the compiler goes up.

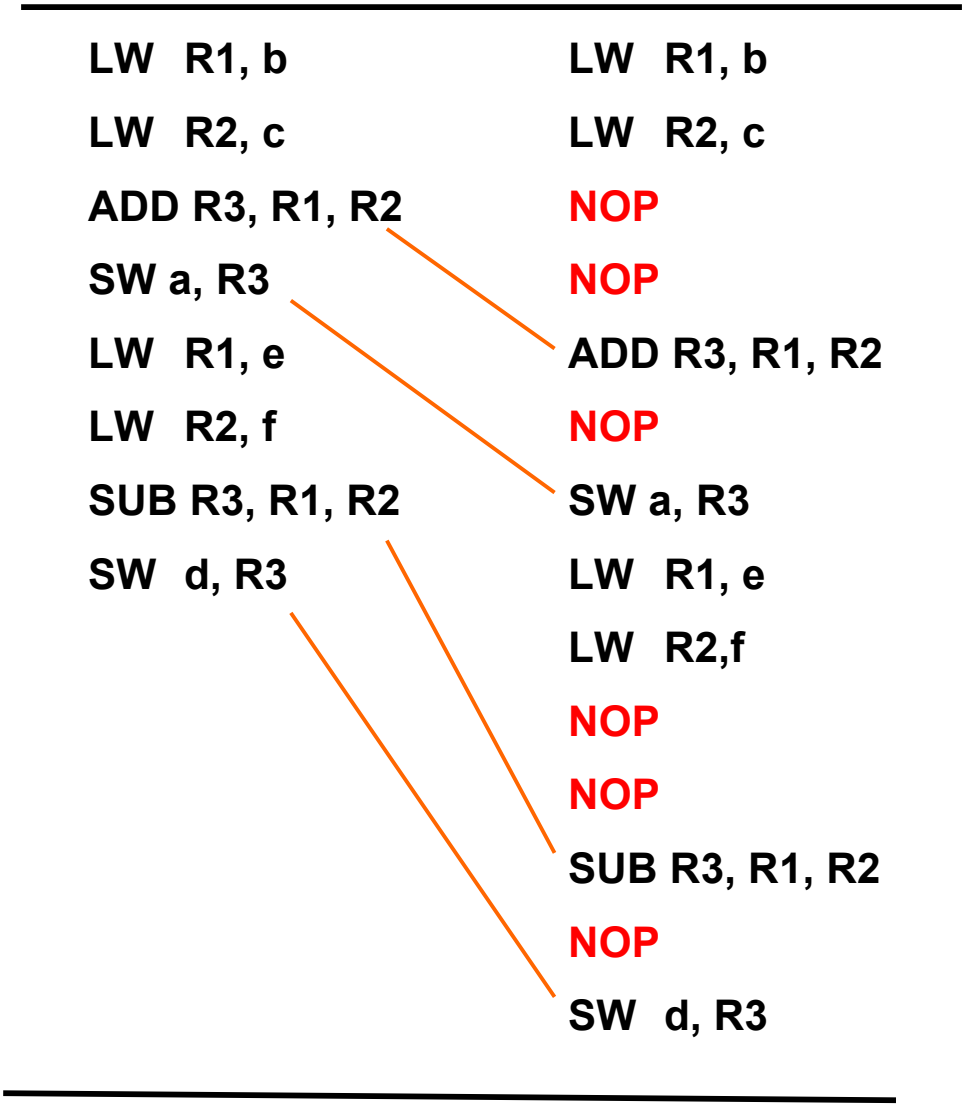
## Instruction Reorganization by Compiler

To avoid data hazards, the control unit can insert bubbles.

As an alternative, the compiler can use **NOP** instructions.

**Example: Compute a: = b + c; d: = e + f**

**(a, b, c, d, e, f** are stored in the memory**)**




*Original code* *Code generated by a smart compiler*

# Instruction Reorganization by Compiler

The compiler can further speedup by reorganizing the instruction stream and minimizing the no of NOP's.

**Example:** **Compute a: = b + c; d: = e + f**

| | |
|---|---|
| **LW R1,b** | **LW R1,b** |
| **LW R2,c** | **L W R2,c** |
| **ADD R3, R1, R2** | **LW R4, e** |
| **SW a, R3** | **LW R5, f** |
| **LW R1, e** | **ADD R3,R1,R2** |
| **LW R2,f** | **NOP** |
| **SUB R3, R1, R2** | **SW a, R3** |
| **SW d, R3** | **SUB R6, R5, R4** |
| | **NOP** |
| | **SW d, R6** |
| | **NOP** |

*Original code* *Code reorganized by a smart compiler*

***(Control unit remains unchanged)***

***Note the reassignment of registers***

# Another example: delayed branch

```
        add $r1, $r2, $r3          add $r1, $r2, $r3
        beq $r2, $zero, L          beq $r2, $zero, L'
        <instruction>              NOP (delay slot)
        <instruction>              <instruction>
L:      <instruction>              <instruction>
                                L': <instruction>
```

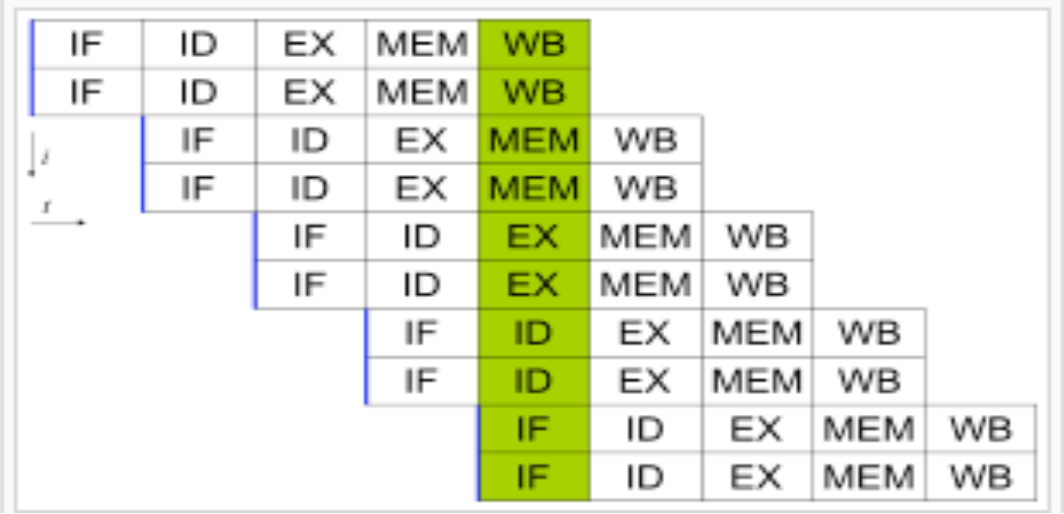
The compiler may try to schedule other instructions in the delay slot for the sake of speed-up. Here is an example:

```
        beq $r2, $zero, L
        add $r1, $r2, $r3
        <instruction>
        <instruction>
L:      <instruction>
```

Note that the first two instructions have been swapped. How does it help?

# Superscalar processors

Helps execute more than one instruction per cycle. Consider as if there are multiple pipelines. Multiple instructions are fetched and issued in a cycle. Also called Second Generation RISC.




Taken from Wikipedia

## Instruction level parallelism

Either the compiler or the control unit must identify which instructions can be executed in parallel. Instruction Level Parallelism (ILP) is a key issue in superscalar design.