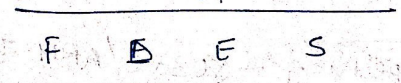
1) Overview of pipelining ↔ instr. pipeliner
2) Pipelined data Path
3) Pipeline hazards
4) Pipeline with interlocks
5) Forwarding
6) Performance metrics
7) Interrupts / Exceptions

→ hardware mech.

Pipelining: Stages



The stages include.
Instruction Fetch (IF)
Instruction decode (ID)
Execute (EX)
Memory (MEM)
Write back (WB)

~~Pipeline hazards~~

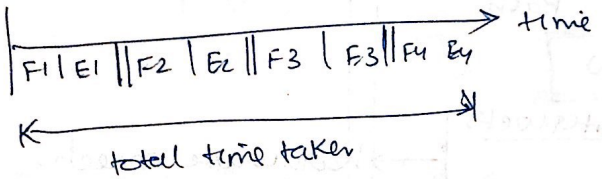
• Pipelining:

- overlapping different stages of an instr is called pipelining.

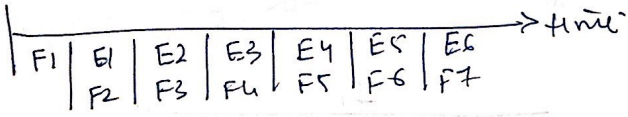
- An instruction requires several steps which mainly involve fetching, decoding and execution. So in to make processor faster a method called pipelining is applied.

* 2 stage Pipelining - 8085

8085 = Non-pipelined processor. (F E F E F E . . —).

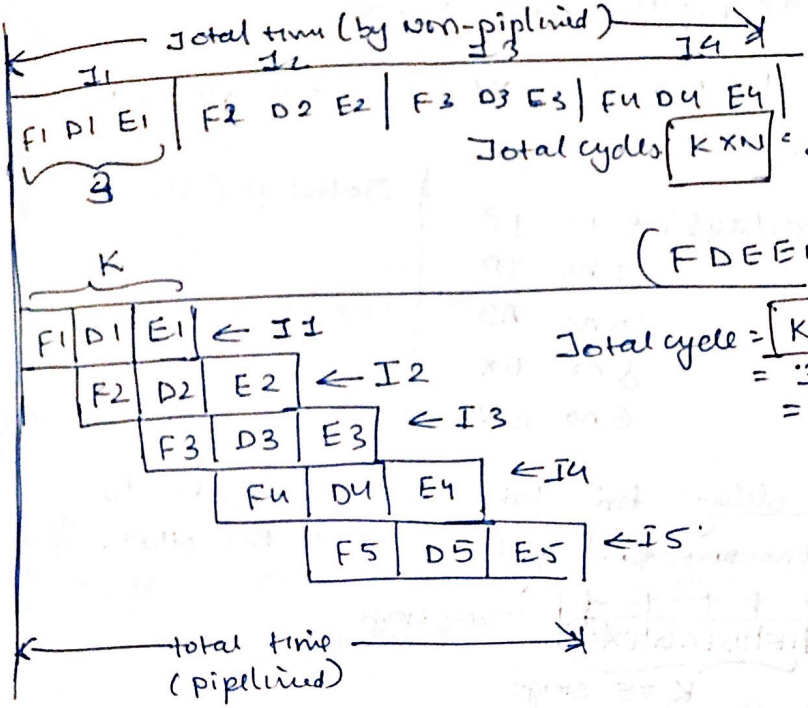


8086 = pipelined processor (F E E E E E . .).



* 3-stage Pipelining - 80386/ARM-7 (P.

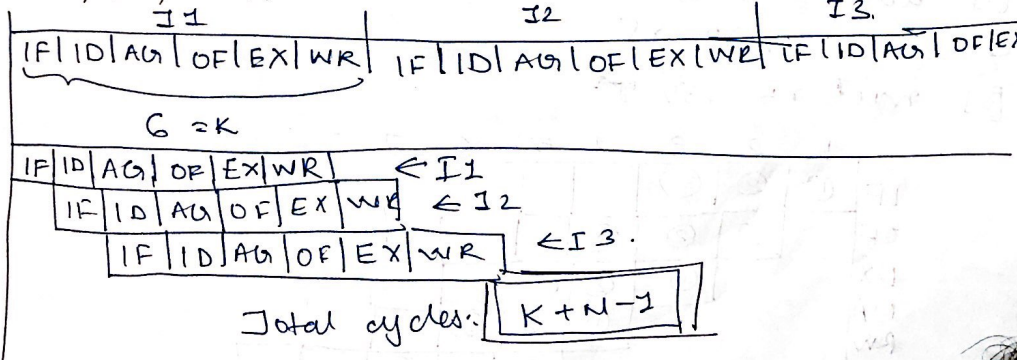
Here in 3 stages the instruction process is divided into 3 stages: 1) Fetching
2) decoding
3) execution.



* 4 stage pipelining
  has includes fetch, Decoder execute and store.

* 5 stage pipelining - PENTIUM
  ↳ Fetch, decoder, Address generation, execute, store

* 6 stage pipelining - Pentium Pro
  IF, ID, AG, OF, EX, WR.



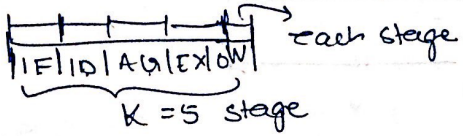
Q Instruction pipeline with 5 stage

to IF, ID, AG, EX, OW without any branch prediction

Total 12 instr

The stage delays: 5 ns IF
4 ns ID
10 ns AG
8 ns EX
6 ns OW

* Propagation delay: time taken for all signals to travel from one end of the stage to the other.



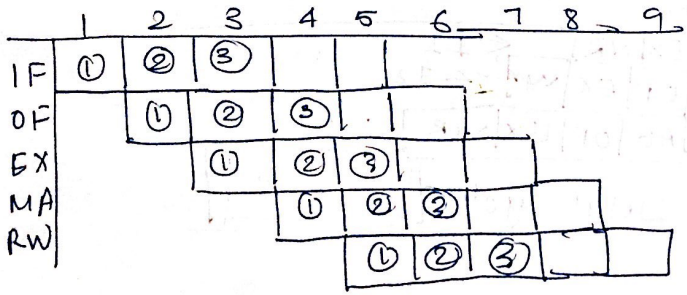
Intermediate storage buffer having ~~delay~~ buffer delay of 1ns.

I4 is the only branch inst, branch target is I9

time needed to complete program?

Pipeline diagram:

[1] add r1, r2, r3
[2] sub r4, r2, r5
[3] mul r5, r8, r9



Instruction Packet

→ Instr contents
→ Program counter
→ intermediate results
→ control signals

~~Data Hazards~~
1) ~~RAW (read after write)~~.

* Pipeline Hazards

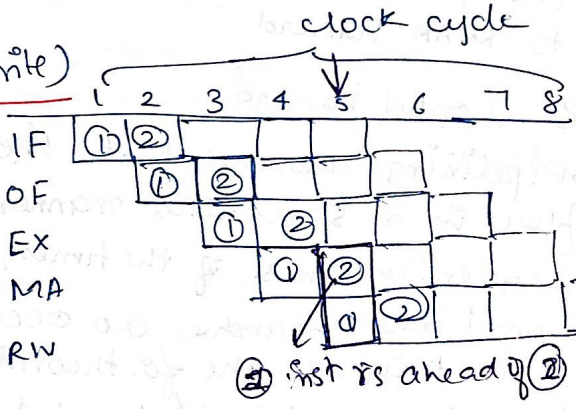
(1). Data hazards / Data dependency hazard

1.1) ~~RAW (Read after write)~~

Data hazard is caused when the result of one instruction becomes the source of the next instruction

1.1) RAW (Read after write)

[1] add (r1), r2 r3
[2] sub r3, (r1) r4



*
IN-ORDER PIPELINE

↳ a preceding instr. is always ahead of a succeeding instr. in the pipeline
→ It includes RAW hazards only.

* out-of order pipelines can have WAR and WAW hazards

- **WAW hazard**

[1]: add r1, r2, r3 r1 ← r2+r3

[2]: sub r1, r4, r3. r1 ← r4-r3

⇒ Instr [2] cannot write the value of r1, before Instr [1] writes to it. will lead to a WAW hazard.

- **WAR hazard**

[1]: add r1, r2, r3 r1 ← r2+r3

[2]: add r2 r5, r6. r2 ← r5+r6

⇒ Instr [2] cannot write the value of r2, before Instr [1] ~~writes to~~ reads the value of r2. leads to WAR hazard.

② **Control hazard**

→ Pipelining assumes that the prgm will always flow in a sequential manner. The programs are sequential most of the times, it is not true always Sometimes, branches do occur in our prgm. So in this, all the forthcoming instructions that have been fetched/decoded have to be flushed/discarded. and the process has to start all over again from the branch add.

[1]: beq .foo

[2]: mov r1, 4

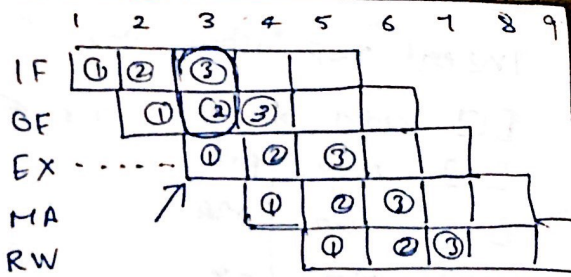
[3]: add r2, r4, r3

...

...

.foo:

[100]: add r4, r1, r2



if branch is taken, then instr [2] and instr [3] might get fetched incorrectly

* Structural hazards

→ caused by physical constraints in the architecture like the buses.

→ caused by diff instr try to access the same resource or

even in the most basic form of pipelining, we want to execute one instr. and fetch the next one now if execution involves registers, pipelining is possible. But if execution requires to read/write data from memory.

[1]: st r4, 20[r5]

[2]: sub r8, r9, r10

[3]: add r1, r2, 10[r3]

[3]: tries to read 10[r3] (MA unit) in cycle 4

[1]: tries to write to 20[r5] in cycle 4.

---

* Solutions in software

- Data hazards.

[1]: add r1, r2, r3

[2]: sub r3, r1, r4,

Insert nop instructions.

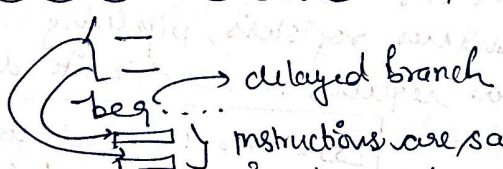
[1]: add r1, r2, r3 ← done
[2]: nop } RW
[3]: nop } MA
[4]: nop } EX
[5]: SUB r3, r1, r4 OF

Reorder the code:

Q
```
add r1, r2, r3                add r1, r2, r3
add r4, r1, 3                 add r8, r5, r6
add r8, r5, r6        →       add r10, r11, r12
add r9, r8, r5                nop
add r10, r11, r12             add r4, r1, 3
add r13, r10, r2              add r9, r8, r5
                              add r13, r10, r2
```

• Solution for control hazard

⇒



Q
```
add r1, r2, r3                b. foo
add r4, r5, r6                add r1, r2, r3   } delay
b. foo                        add r4, r0, r6   }
add r8, r9, r10.              add r8, r9, r10
```

if there is no valid inst b4 branch inst then,
pt inserts [nops].

HARDWARE MECHANISM

Interlocks:

→ Type of hardware mechanism to enforce correctness (interlock).

Two kinds of interlocks

1). Data-Lock

→ Do not allow the consumer instr to move beyond the OF stage till it has reached the correct values.

Implication: Stall the IF and OF stages.

2). Branch Lock

→ we never execute instr. in the wrong paths.

Pipeline bubble → nop inst

↳ it is inserted into a stage when the prev stage needs to be stalled.

Q How to stall a pipeline?

1. Disable the write functionality of

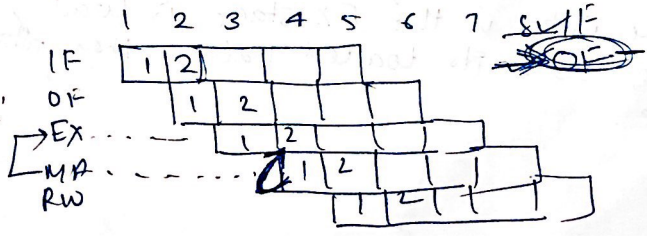
→ IF-OF register

PC (program counter)

2 to insert a bubble.

→ write a bubble (nop instr) into the OF-EX register.

Forwarding: (No bubbles)



Q ~~when does inst~~

When does inst 2 need the value of r1?

→ cycle 3, OF stage ✗

cycle 4, EX stage

Q. When does I1 produces the value of r1?

→ cycle 5, RW stage ✗

End of cycle 3, EX stage

Types of Forwarding paths

* 3 stage path

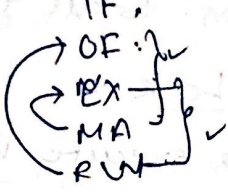
→ RW → OF

* 2 Stage path

RW → EX

~~MA → EX~~

MA → OF (Not required)



* 1 Stage path.

MA → EX (ALU, load, store)

~~MEX~~ → OF (X)

RW → MA (load to store)

*.

Interlocks with forwarding

1.) Data Lock.

→ Only need to check for the load-use hazards

→ If the instr. in the EX stage is load and OF stage uses its loaded value - then stall for 1-cycle.

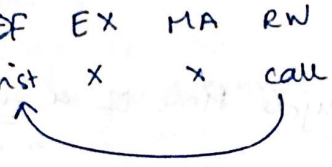
2) Branch - Lock.

→ Remains Lock.

* The curious case of call instruction

→ The call instruction generates the value of ra in the RW stage

→ Any instruction that uses the value of ra (such as ret) still works.



* Performance metrics:

→ Performance is inversely proportional to the time it takes to execute a program.

$T = \#seconds$

$= \frac{\#seconds}{\#cycle} \times \frac{\#cycle}{\#instr} \times \#instr$

$\frac{\#seconds}{\#cycle}$: time $\frac{1}{f}$

$\frac{\#cycle}{\#instr}$: CPI

CPI : cycles per instr.

f : frequency (cycles / sec)

$$T = \frac{CPI \times \#instr}{f}$$

$$P \propto \frac{IPC * f}{\#instr}$$

→ performance

* Computing the CPI

n instr, K stages..

$$CPI = \frac{n + K - 1}{n} \quad \begin{matrix} \leftarrow \text{cycles} \\ \leftarrow \text{instr} \end{matrix}$$

* max frequency..

t_{max} = maximum amount of time that it takes to execute any instr in algo work.

minimum clock cycle time of a single cycle pipeline = t_{max}

If the pipeline has stages and balanced then

Time per stage = t_{max}/k

If stages are not balanced, having some delay.

latch delay = l

$$t_{stage} = \frac{t_{max}}{K} + l$$

$$\frac{1}{f} = \frac{t_{max}}{K} + l$$