Computer Architecture
4. CPU Pipeline

Lecturer: A.Prof.Dr. Hoàng Xuân Dậu

Email: dauhx@ptit.edu.vn

Faculty of Information Security

Posts & Telecommunications Institute of Technology

Main topics

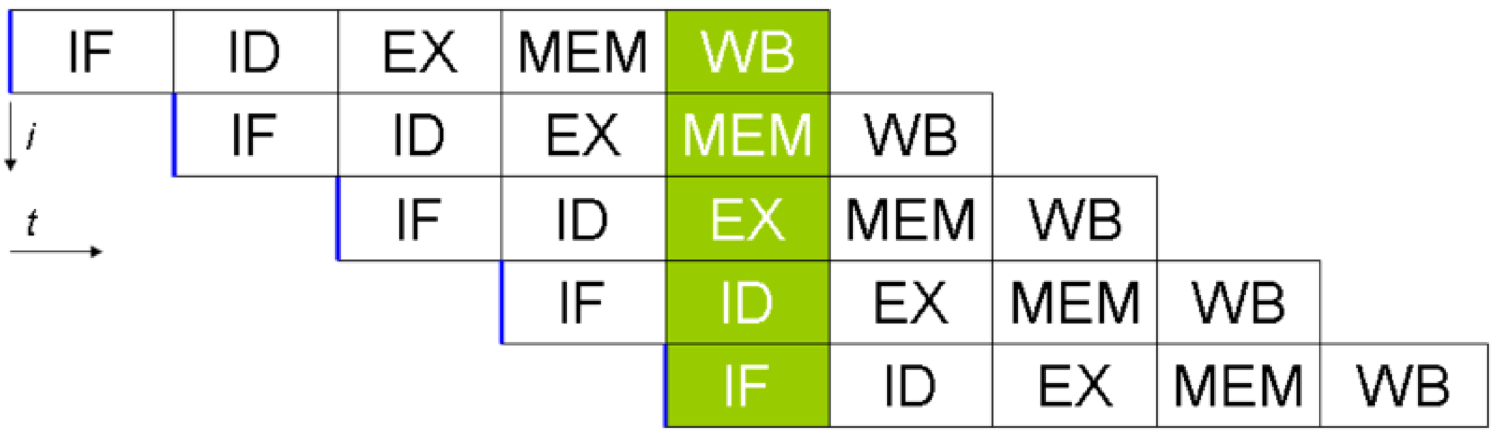
- Introduction to CPU pipeline
- Pipeline issues
- Handling data & resource conflicts
- Handling branching
- Some CPUs’ pipelines
- Super pipeline

Pipeline – Automobile Assembly

- A assembly line is divided into number of stages;
- There are several cars are in the line;
- One task is done in a stage;
- In a certain period of time, a complete car is out and another raw car gets in.

Pipeline – Basic Principles

Without pipeline

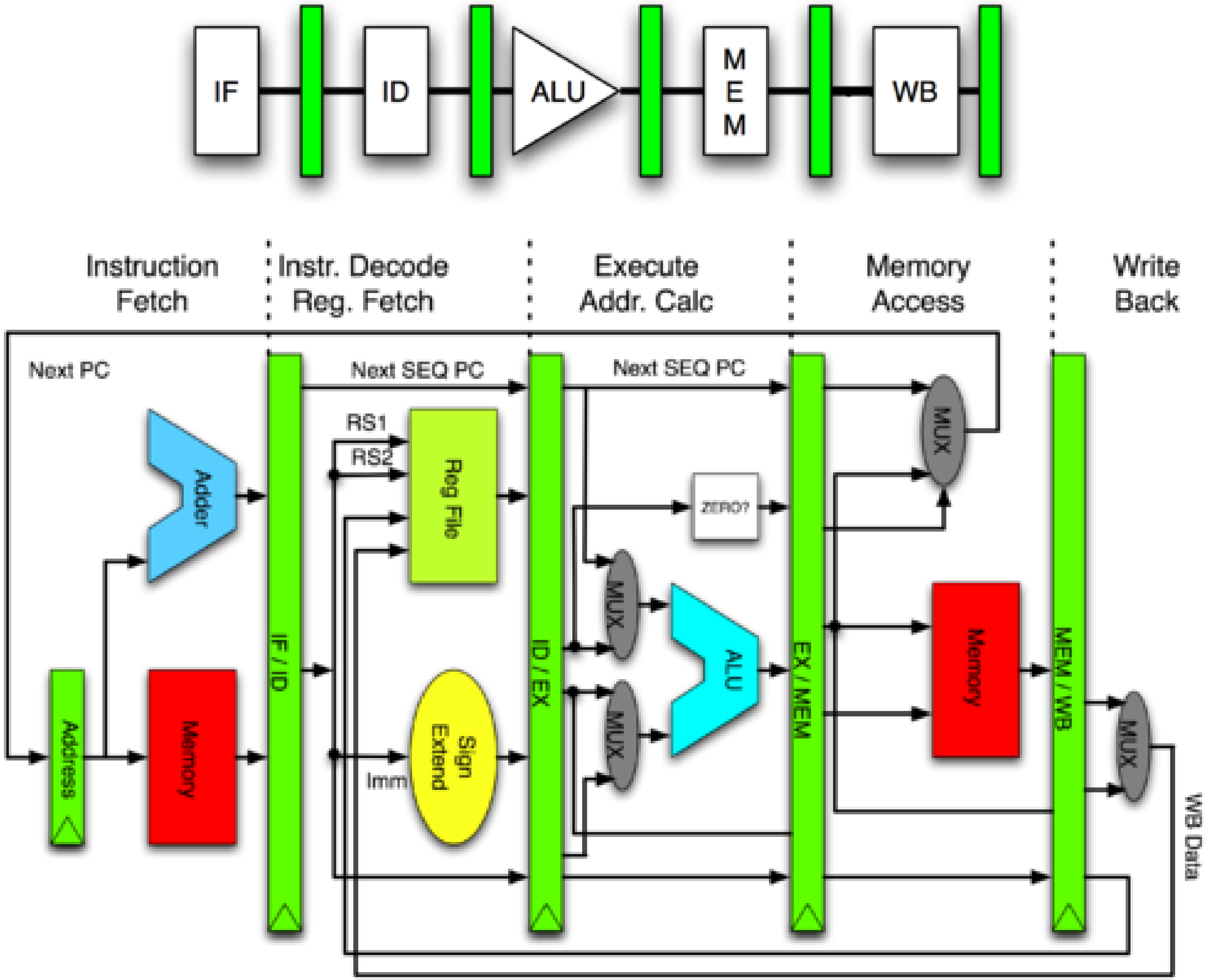


With pipeline

Pipeline – Basic Principles

- The instruction execution is divided into stages
- 5 stages of a typical load-store system:
  - Instruction Fetch - IF: Fetch instruction from memory (or cache)
  - Instruction Decode - ID: decode instruction and fetch operands
  - Execute - EX: execute instruction; if it is memory access instruction: compute memory address
  - Memory Access - MEM: Read/write memory; no-op if not access memory
  - Write Back - WB: store result into registers.

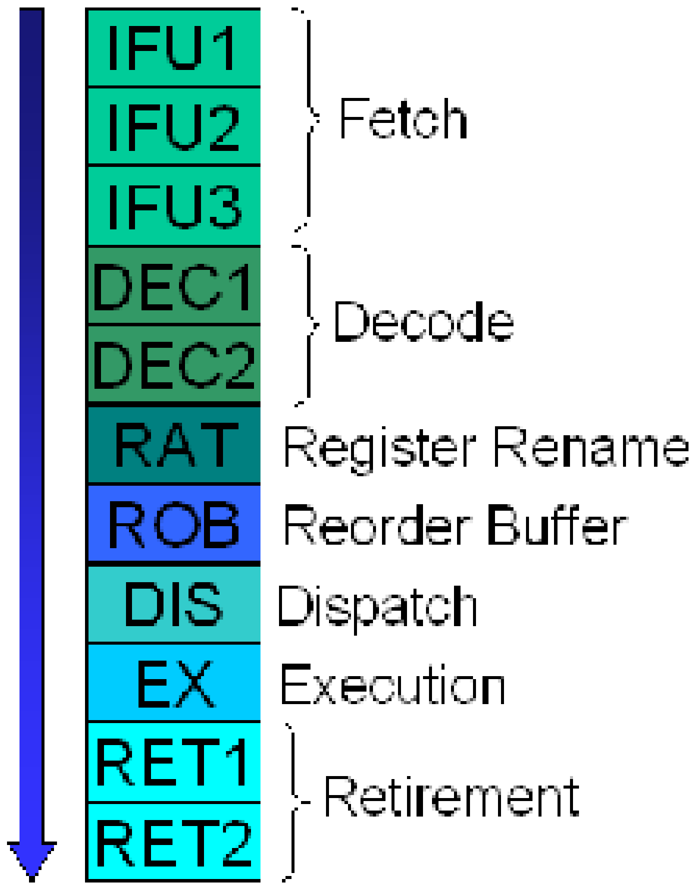
Pipeline – Basic Principles



Pipeline - Characteristics

- Pipeline is the instruction level parallelism (ILP);
- A pipeline is *complete* if it always accepts a new instruction at each clock cycle;
- A pipeline is not *complete* if there are delayed stages in its execution;
- The number of pipeline stages depends on CPU design:
  - 5 stages: simple pipelines
  - 10-15 stages: Pen III, M, Core 2 Duo, Core i_3, Core i_7
  - 20 stages: Pen IV (except Prescott)
  - 31 stages: Pen IV Prescott

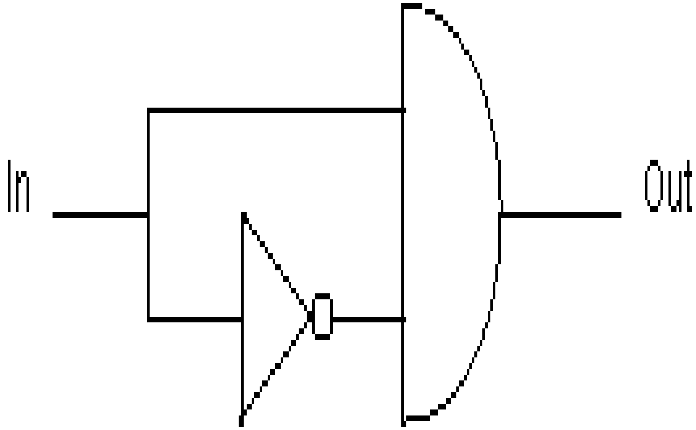
Pipeline – P6 (Pen III, M)



Select number of stages

- Stage execution time
  - All stages should have the same execution time;
  - Slow stages should be divided.
- Resource issues
  - What happens when IF and ID stages of two instructions in pipeline are accessing memory at the same time?
- Select number of stages
  - In theory, the more number of stages the better performance;
  - If a long pipeline is empty due to some reasons, it will take more time to fill the pipeline.

Pipeline Hazards



Logic Gate Hazard

- The expected output is always 0 (false)
- But at some cases, the output is 1 (true) → Hazard.

Pipeline Hazards/Issues

- Resource conflicts
- Data conflicts (mostly RAW or Read After Write Hazards)
- Branch instructions

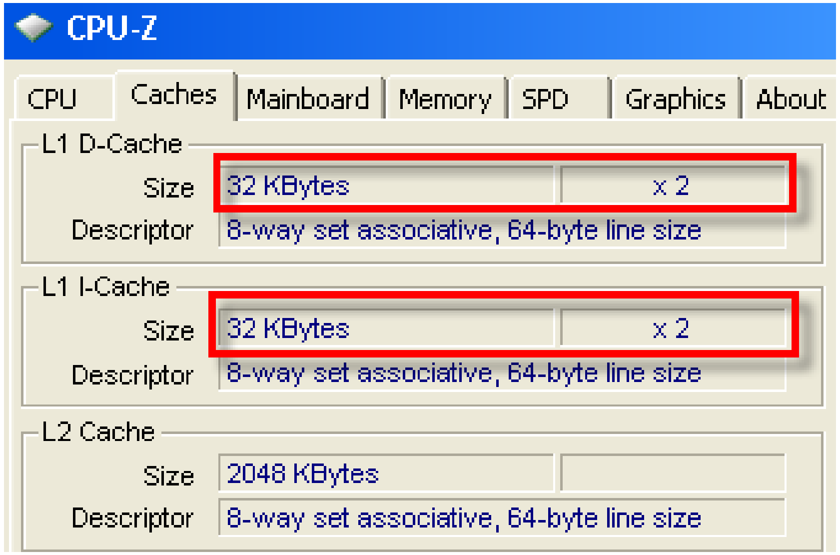
Pipeline Hazards - Resource conflicts

- Resources are not sufficient
- Example: if memory only supports one read/write operation at a time, simultaneous memory accesses from pipeline will cause resource conflicts.

IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

Pipeline Hazards - Resource conflicts

- Solutions:
  - Improve resource capabilities
  - Memory/Cache: supports more read/write operation at a time
  - Split L1 cache into I-Cache and D-Cache to improve concurrent accesses.



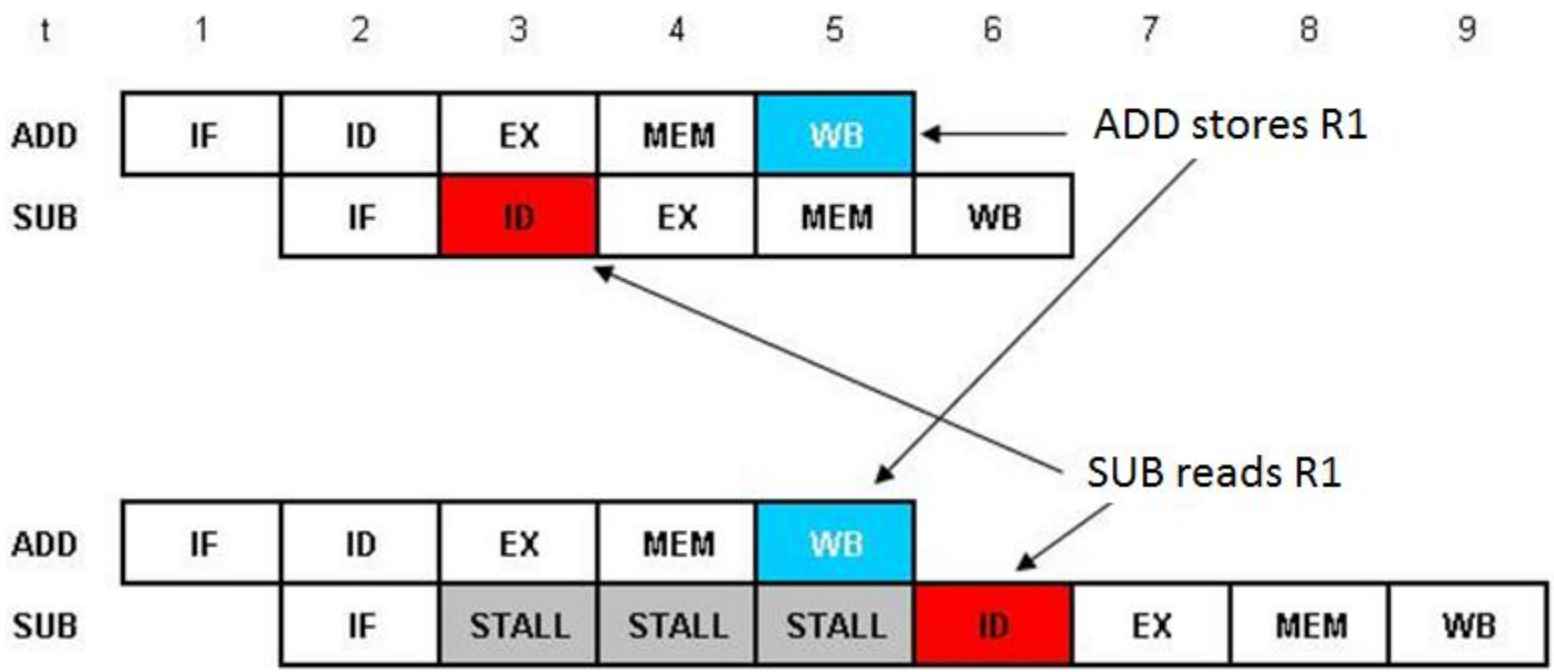
Pipeline Hazards - Data conflicts

- Consider two instructions:

  ADD R1, R1, R3 ; R1 ← R1+R3

  SUB R4, R1, R2 ; R4 ← R1-R2
- SUB uses ADD’s result – there are data dependency between these 2 instructions.
- SUB reads R1 at stage 2 (ID), while ADD stores result at stage 5 (WB)
  - SUB reads R1’s old value before ADD stores the R1’s new value.

Pipeline Hazards - Data conflicts



- ADD R1, R1, R3 ;R1 ← R1+R3
- SUB R4, R1, R2 ;R4 ← R1-R2

Data conflicts – Possible Solutions

t	1	2	3	4	5	6	7	8	9
ADD	IF	ID	EX	MEM	WB				
NO-OP		NO-OP	NO-OP	NO-OP	NO-OP	NO-OP			
NO-OP			NO-OP	NO-OP	NO-OP	NO-OP	NO-OP		
NO-OP				NO-OP	NO-OP	NO-OP	NO-OP	NO-OP	
SUB					IF	ID	EX	MEM	WB

Delay SUB execution by inserting 3 NO-OP

Data conflicts – Possible Solutions

- Identify it happening
- Stall the pipeline: we must delay or stall the pipeline using some methods until the correct data value is available
- Use compiler to identify RAW and:
  - Insert NO-OP instructions into 2 instructions that have RAW;
  - Re-order the program and insert data independent instructions into the position between 2 instructions that have RAW.
- Use hardware components to identify RAW (implemented in modern CPUs).

Data conflicts – Possible Solutions

t	1	2	3	4	5	6	7	8	9
ADD R1, R1, R3	IF	ID	EX	MEM	WB				
LOAD R4, [1000]		IF	ID	EX	MEM	WB			
ADD R5, 500			IF	ID	EX	MEM	WB		
STORE [2000], R6				IF	ID	EX	MEM	WB	
SUB R4, R1, R2					IF	ID	EX	MEM	WB

Insert 3 data independent instructions into the position between ADD and SUB

Management of Branching Instructions

- The portion of branching instructions is about 30%. Branching instructions would make:
  - Interruption in the normal execution of program
  - Pipeline empty if there are no advance prevention measures.
- For CPUs with long pipeline (P4 with 31 stages), the branching issue is more complicated because:
  - Push all being-executed instructions out of the pipeline
  - Reload new instructions into pipeline

Management of Branching Instructions

t	1	2	3	4	5	6	7	8	9
JUMP [1000]	IF	ID	EX	MEM	WB				
LOAD R4, [00FF]		IF	ID	EX	MEM	WB			
ADD R5, 500			IF	ID	EX	MEM	WB		

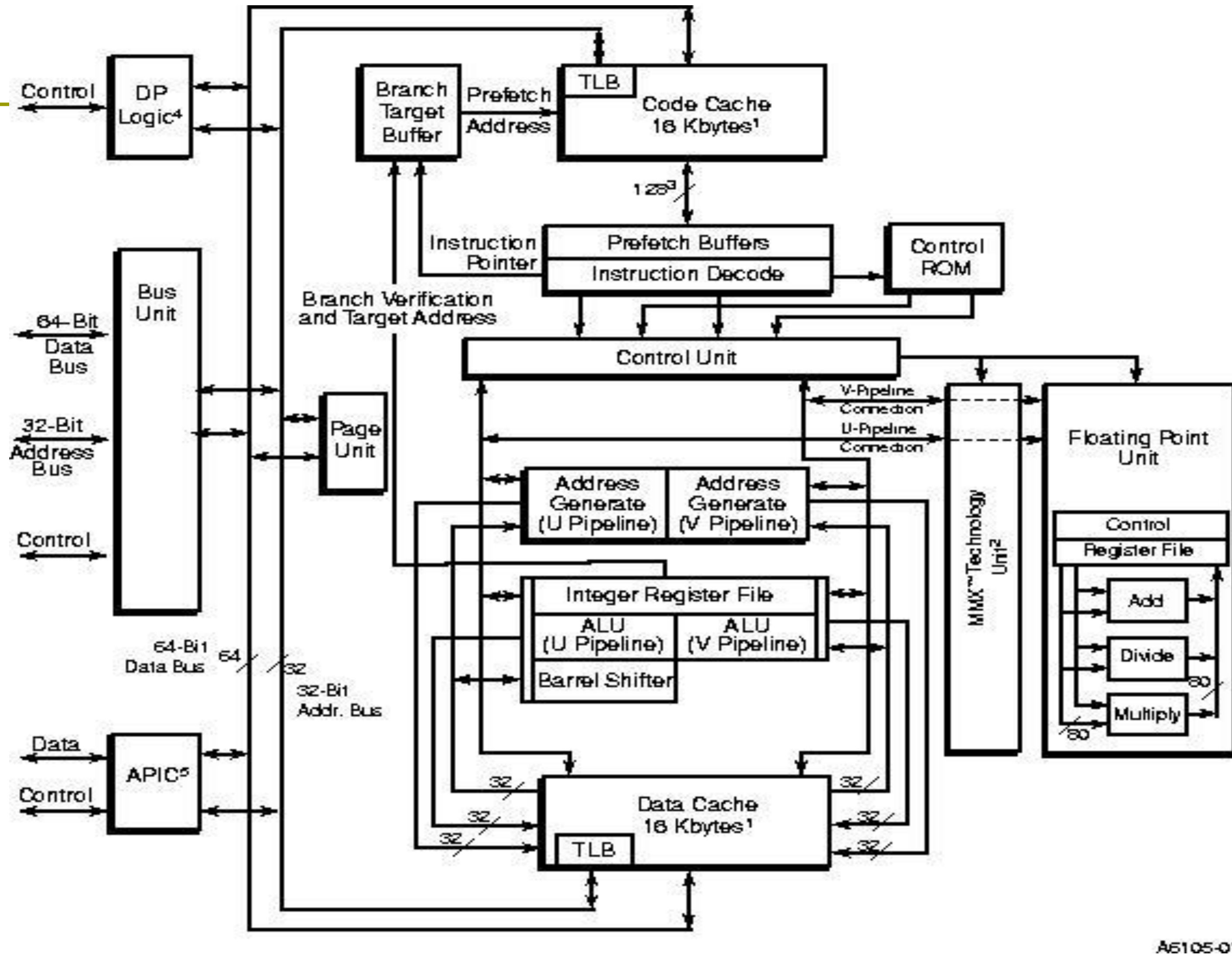
t	1	2	3	4	5	6	7	8	9
JUMP [1000]	IF	ID	EX	MEM	WB				
1000: ADD R6, 2000				IF	ID	EX	MEM	WB	
LOAD R7, [03FF]					IF	ID	EX	MEM	WB

When a branching instruction is executed next instructions are pushed out of pipeline and new instructions at branching target are loaded

Branching Instructions - Solutions

1. Branch Targets
2. Conditional Branches
   - Delayed Branching
   - Branch Prediction

Branch Targets in Pentium III



Branch Targets

- When a branching instruction is executed, the next instruction to be fetched is the one at branch target, not the instruction at next position.

```
JUMP  <Address>
ADD R1, R2


Address: SUB R3, R4
```

Branch Targets

- Branching instructions are identified at the ID stage and can be done by pre-decode;
- Use *Branch Target Buffer* (BTB) to store:
  - Target addresses of executed branching instructions;
  - Target instruction of executed branching instructions;
- If these branching instructions are re-used (especially in a loop):
  - Their target addresses stored in BTB can be used without re-calculation
  - Target instructions can be used directly without fetching again from memory.

  → this is possible because target address and instruction are normally unchanged.

Conditional Branches

- It is more difficult to manage conditional Branches because:
  - There are 2 target instructions to select
  - It is not possible to determine the target instruction until the branching instruction is executed
  - The use of BTB is not efficient because we have to wait until we can determine the target instruction.

Conditional Branches - Strategies

- Delayed branching
- Branch prediction

Delayed branching

- It is based on the idea that
  - branch instruction will not cause immediate branching
  - but it will be “delayed” some clock cycles depending on the length of pipeline.
- Delayed branching characteristics:
  - Work well on RISC processors in which instructions have similar execution time;
  - Short pipelines (normally 2 stage pipelines)
  - The instruction after the branch instruction is always executed no matter what the branch instruction’s result is.

Delayed branching

- Implementation:
  - Use compiler to insert NO-OP into the position right after the branching instruction, or
  - Move an independent instruction from before to right after branching instruction.

Delayed branching

- Consider instructions:

```
ADD R2, R3, R4
CMP R1,0
JNE somewhere
```

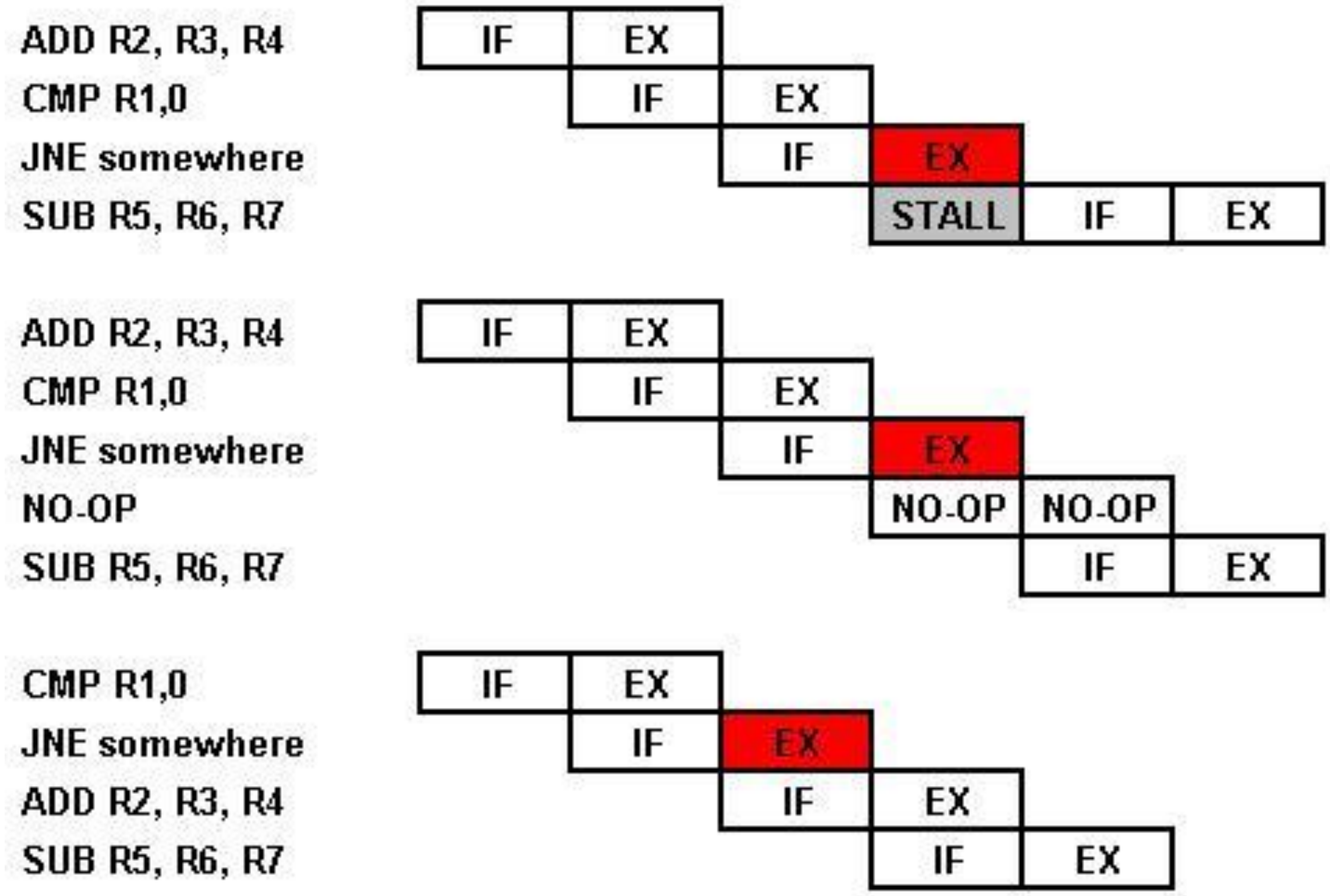
- Insert NO-OP into the position right after the branching instruction:

```
ADD R2, R3, R4
CMP R1,0
JNE somewhere
NO-OP
```

- Move an independent instruction from before to right after branching instruction:

```
CMP R1,0
JNE somewhere
ADD R2, R3, R4
```

Delayed branching



Delayed branching - Comments

- Easy to implement via compiler optimization
- No special hardware are required
- Insertion of NO-OPs would downgrade the pipeline performance
- Replacing NO-OPs by independent instructions can reduce the necessary number of NO-OPs up to 70%.

Delayed branching - Comments

- Increase the code complexity
- Require that programmers and compiler builders have deep knowledge about processor pipelines. This is very difficult to achieve.
- Reduce the portability of the program code because programs have to be re-written or re-compiled on a new processor platforms.

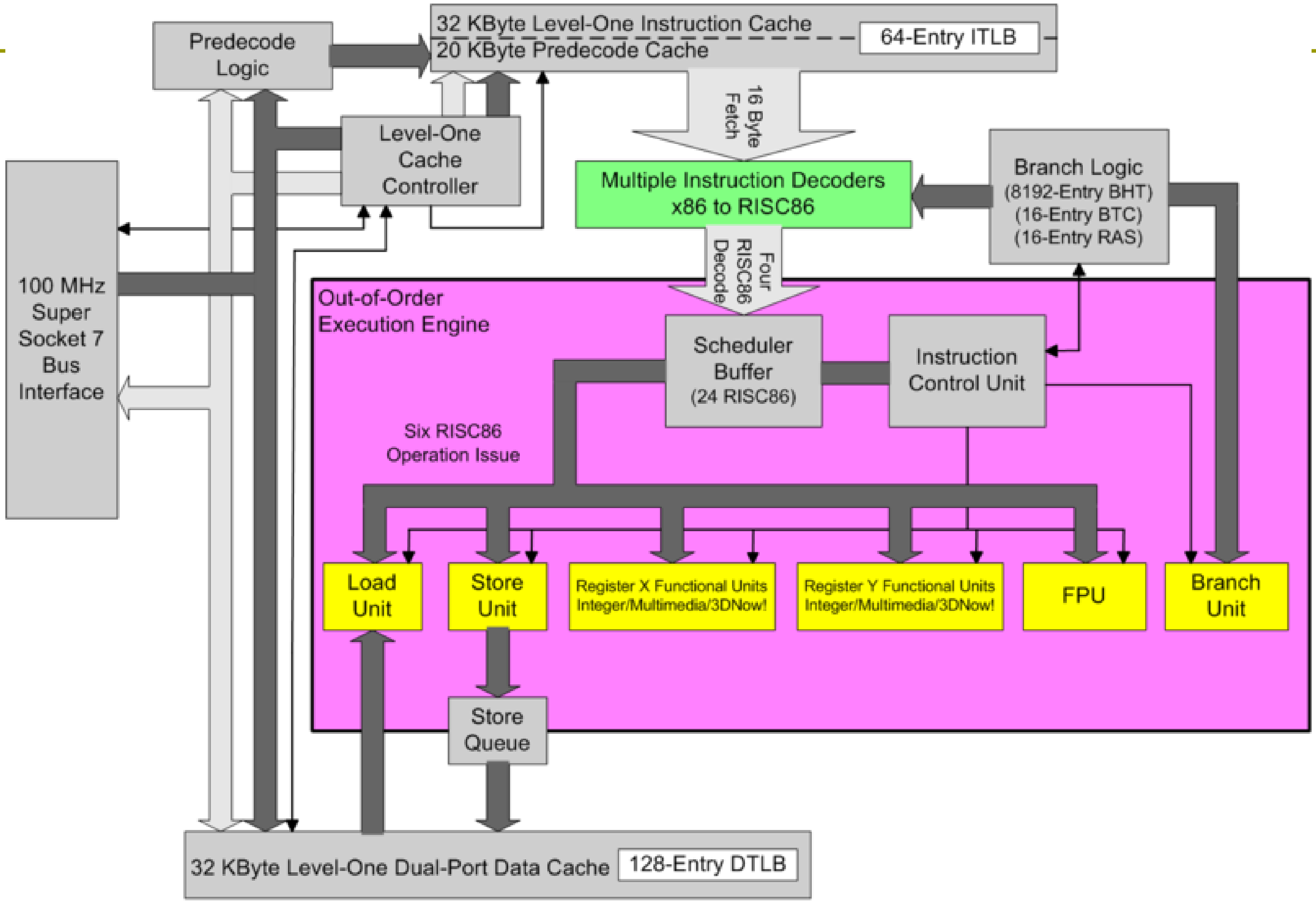
Branch Prediction

- It is possible to predict the target instruction of branching instruction.
  - Correct prediction: improve performance
  - Incorrect prediction: push out next loaded instructions and reloaded instructions at the branch target.
  - The worse case of prediction is 50% correct and 50% incorrect.

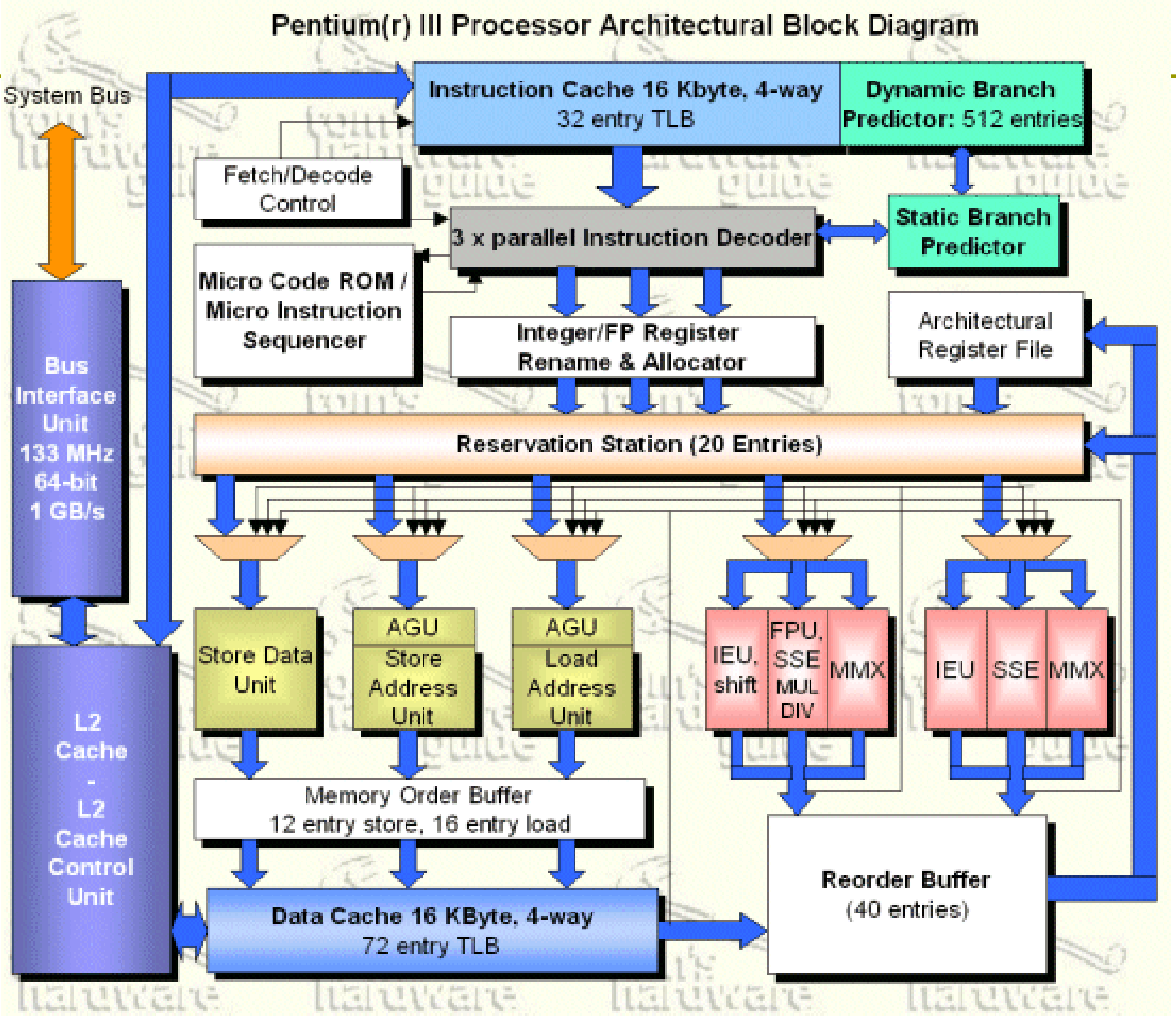
Branch Prediction

- Prediction bases:
  - For backward branch instructions:
    - It is usually a part of a loop
    - Loop is usually executed many times
  - For forward branch instructions:
    - May be the exit of a loop
    - May be a conditional jump

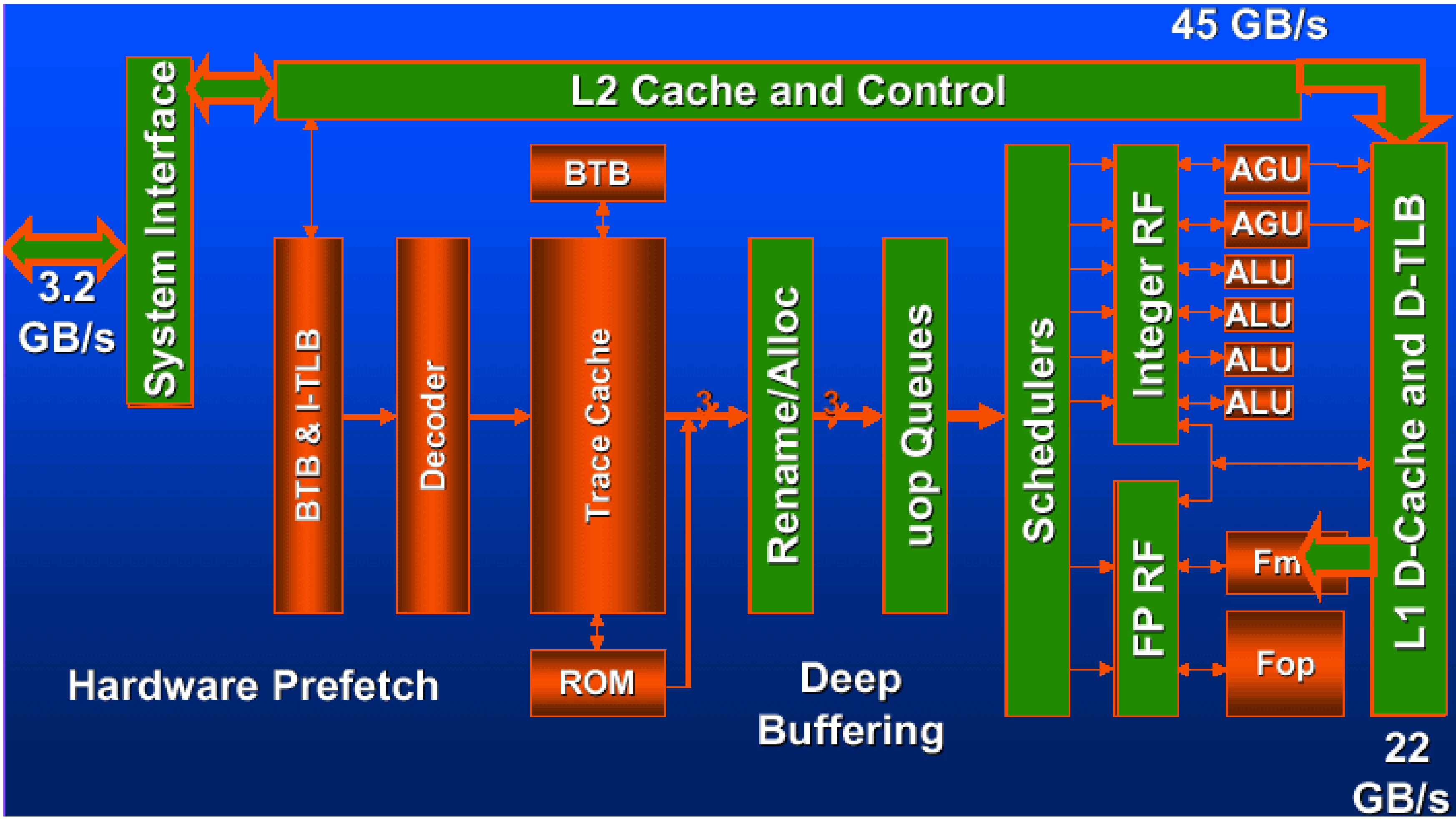
CPU Pipelines – AMD K6-2



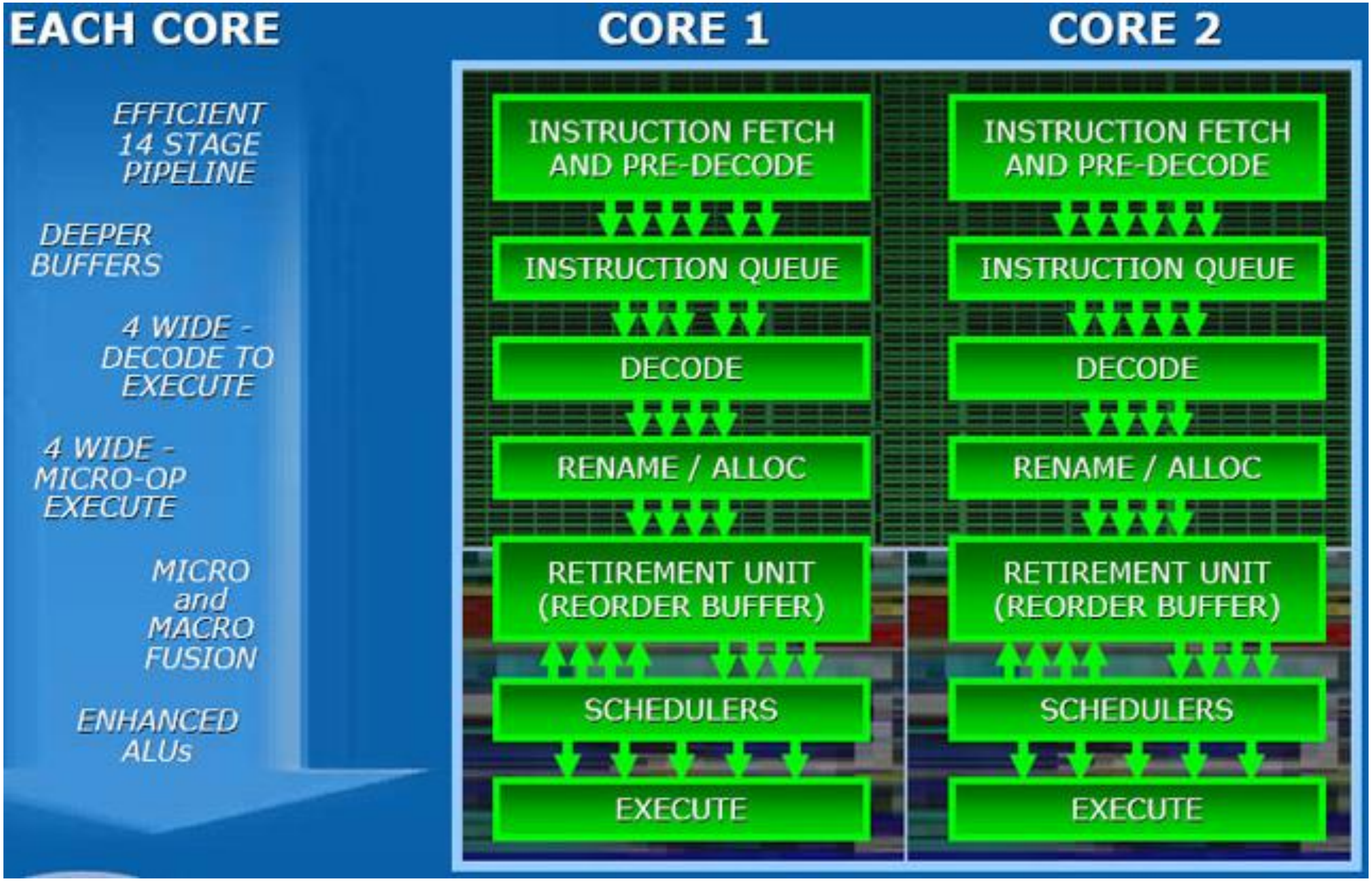
CPU Pipelines – Intel PIII



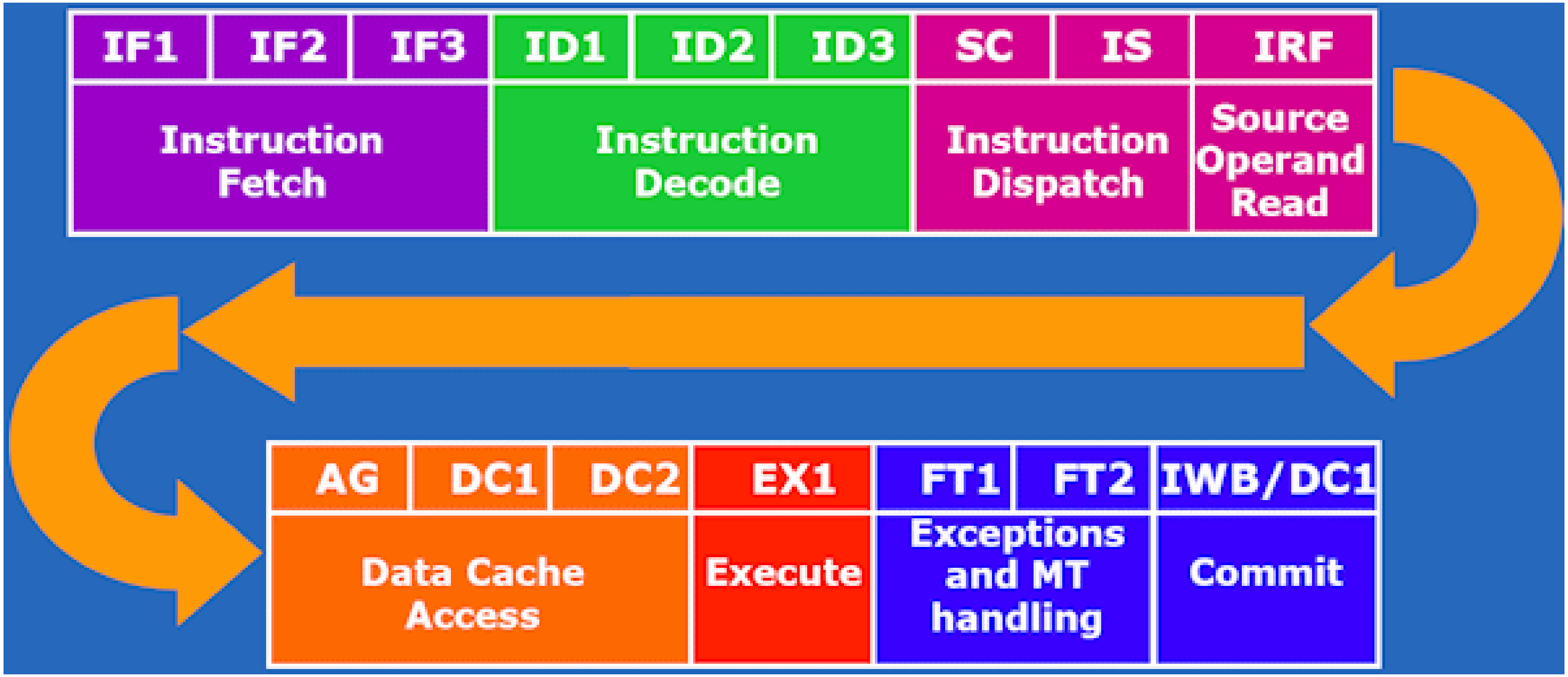
CPU Pipelines – Intel Pentium 4



CPU Pipelines – Intel Core 2 Duo



CPU Pipelines – Intel Atom 16-stage



CPU Superpipelining

- Superpipelining is a technique that allows:
  - Increase the depth of the CPU pipeline
  - Increase CPU clock speed
  - Reduce the delay for each stage of instruction execution.
- Example: If instruction execution stage by ALU is too long, it is divided to some smaller stages and then it helps to reduce the delayed time for short execution stages.

CPU Superpipelining

- Intel Pentium 4 with 20-stage super-pipeline:

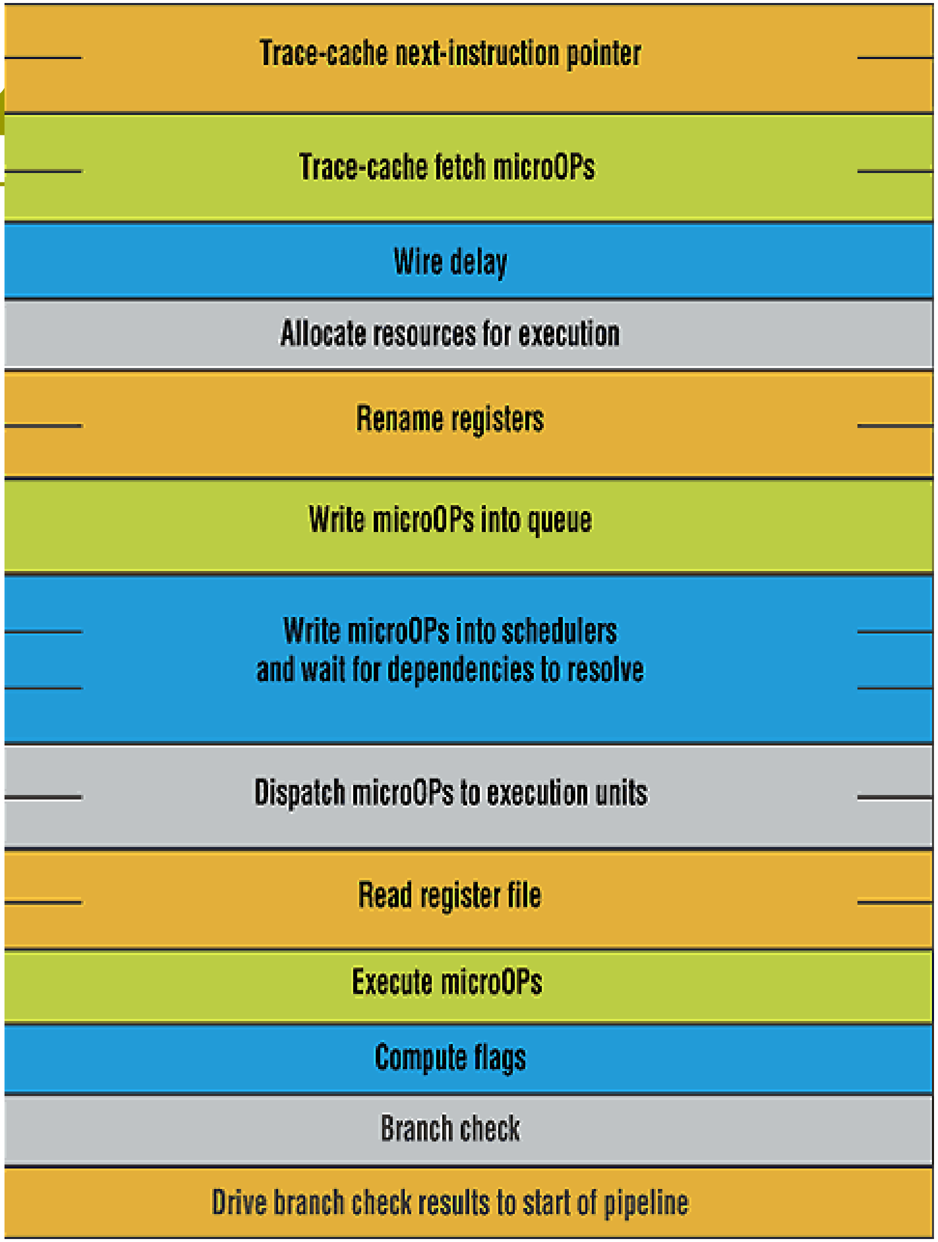
TC	NI	TR	F	D	AR	AR	AR	Q	S	S	S	D	D	R	R	E	F	BC	D			
	TC	NI	TR	F	D	AR	AR	AR	Q	S	S	S	D	D	R	R	E	F	BC	D		
		TC	NI	TR	F	D	AR	AR	AR	Q	S	S	S	D	D	R	R	E	F	BC	D	
			TC	NI	TR	F	D	AR	AR	AR	Q	S	S	S	D	D	R	R	E	F	BC	D

i

t

CPU Superpipeli

- Intel Pentium 4 with 20-stage super-pipeline:



CPU Superpipelining

- Cons of CPU pipelines with to many stages:
  - It will cause longer delay to solve the issues with pipeline hazards:
    - Resouce confícts
    - Data confícts.