COMPUTER ARCHITECTURE

Code: CT173

Part V: Instruction Pipelining

MSc. NGUYEN Huu Van LONG

Department of Computer Networking and Communication,

College of Information & Communication Technology,

CanTho University

Agenda

- Parallelism in computer architecture
  - Concept of design
  - Classification
    - Instruction level parallelism
    - Thread level parallelism
    - Data level parallelism
    - Request level parallelism
  - Instruction level parallelism ILP
    - Dynamic Parallelism
    - Static Parallelism
- Pipelining
  - Concept of design
  - Structure Hazard
  - Data Hazard
    - Instruction Scheduling
    - Data forwarding
  - Control Hazard
    - Static Branch Prediction
    - Dynamic Branch Prediction
- Superpipelining

Parallelism: Concepts of design

- Although computers keep getting faster, the demands placed on them are increasing at least as fast.
- For many users, however much computing power is available, especially in science, engineering, and industry, it is never enough → *John Von Neumann processor is not good enough to satisfy all of these demands* → need a solution to enhance the performance of processor.
- There are several annoyed issues appear when trying to achieve the goal
  - Circuit speed → limited speed of electron and photon
  - Heat dissipation → turn supercomputers into state-of-the-art air conditioners
  - Transistor sizes continue to shrink → quantum-mechanical effects, i.e. Heisenberg uncertainty principle → major problem
  - Sequential Workflow → instruction and data is process one after one in order

Parallelism: Classification

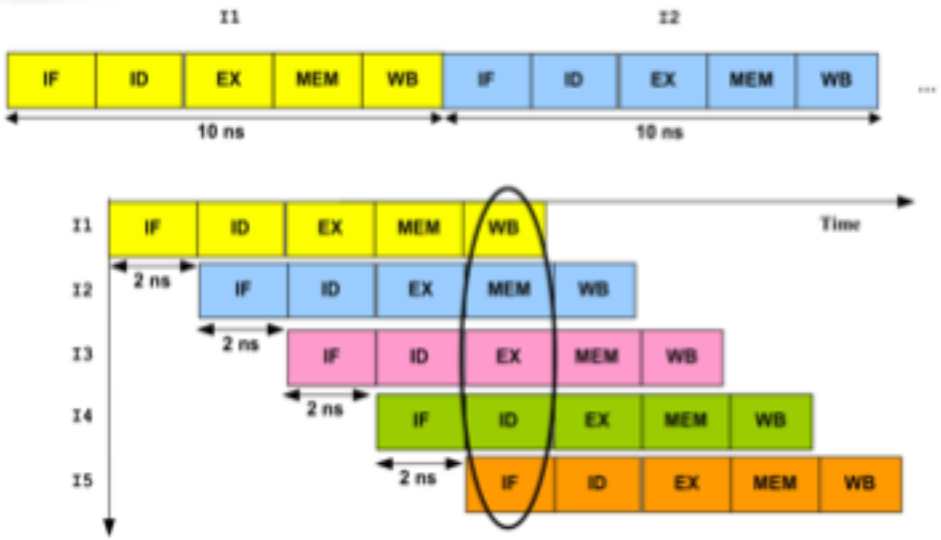
- There are several level of parallelism in computer system design:
  - *Instruction Level Parallelism ILP*: could be done by Pipeline, Superscalar, Very long Instruction Word VLIW
  - *Thread Level Parallelism TLP*: could be done by Multithreading or Hyper threading
  - *Data Level Parallelism DLP*: could be done by GPU, Multiprocessor, Multicomputer or Computer Network
  - *Request Level Parallelism RLP*: could be done by Map Reduce (Hadoop)
- In other way, parallelism could be divided into various hardware level, from bottom to top of design
  - *Processor parallelism*: implemented with ILP, TLP/DLP
  - *Silicon chip parallelism*: implemented with TLP/DLP
  - *Co-processor parallelism*: implemented with DLP
  - *System parallelism*: implemented by DLP and RLP

Instruction Level Parallelism - ILP

- One of the main design issues for today's high-performance microprocessors is exploiting a high level of **Instruction Level Parallelism ILP**
- There are two approaches to ILP
  - ***Dynamic Parallelism*:** depend on the hardware work, the processor decides *at run time* which instructions to execute in parallel, i.e. Pentium
  - ***Static Parallelism*:** depend on compiler work, the compiler decides *at compile time* which instructions to execute in parallel, i.e. Titanium
- Goal of ILP design is to identify and take advantage of as much instructions as possible: ILP allows the compiler or the processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed
- Micro-architectural techniques that are used to exploit ILP include:
  - Instruction pipelining
  - Superscalar, VLIW, Explicitly Parallel Instruction Computing EPIC
  - Out of order execution
  - Register renaming and Speculative execution
  - Branch prediction

Pipelining technique

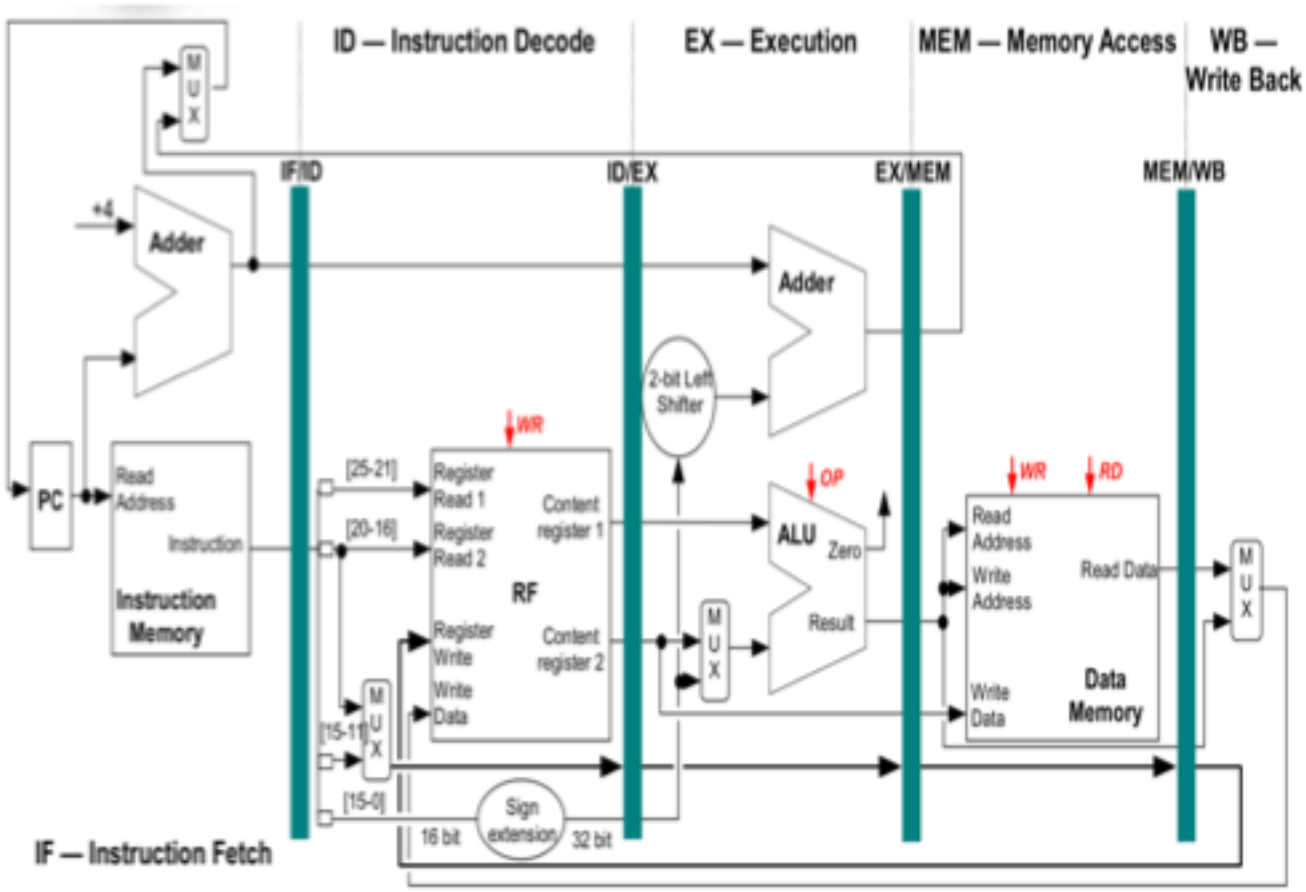
- Performance optimization technique based on the overlap of the execution of multiple instructions deriving from a sequential execution flow
- Pipelining technique is transparent for the programmer.
- It allows faster CPU throughput but may increase latency due to the added overhead of the pipelining process
- Basic idea:
  - The execution of an instruction is divided into different phases **(pipelines stages)**
  - Each stage need a given clock cycle (normally 1 cycle) to proceed in pipeline
  - The pipeline stages should be **synchronized** to balance the length of each pipeline stage
  - In case the pipeline stages are perfectly balanced, the **ideal speedup** due to pipelining is equal to the number of pipeline stages



- **Ex 1**: The single cycle unpipelined *CPU1* with clock cycle of *8ns* and the pipelined *CPU2* with 5 stages of 2 ns.
  - The **latency** of each instruction is worsened: from *8ns* to *10 ns*
  - The **throughput** is improved of 4 times: 1 instr completes in 8ns vs 1 instr completes in 2ns
- **Ex 2**: The multi cycle unpipelined *CPU3* composed of 5 cycles of *2ns* and the pipelined *CPU2* with 5 stages of 2 ns.
  - The **latency** of each instruction is not varied, 10ns
  - The **throughput** is improved of 5 times: 1 instr completes in 10ns vs 1 instr completes in 2ns

Pipeline: Datapath in MIPS64

- We need to pass throughout the **pipeline registers** (*ID/EX*, *EX/MEM* and *MEM/WB*), the address of the register to write during the WB stage.
- The address of the write register pass throughout the pipeline registers and then come from the MEM/WB pipeline register with the write data

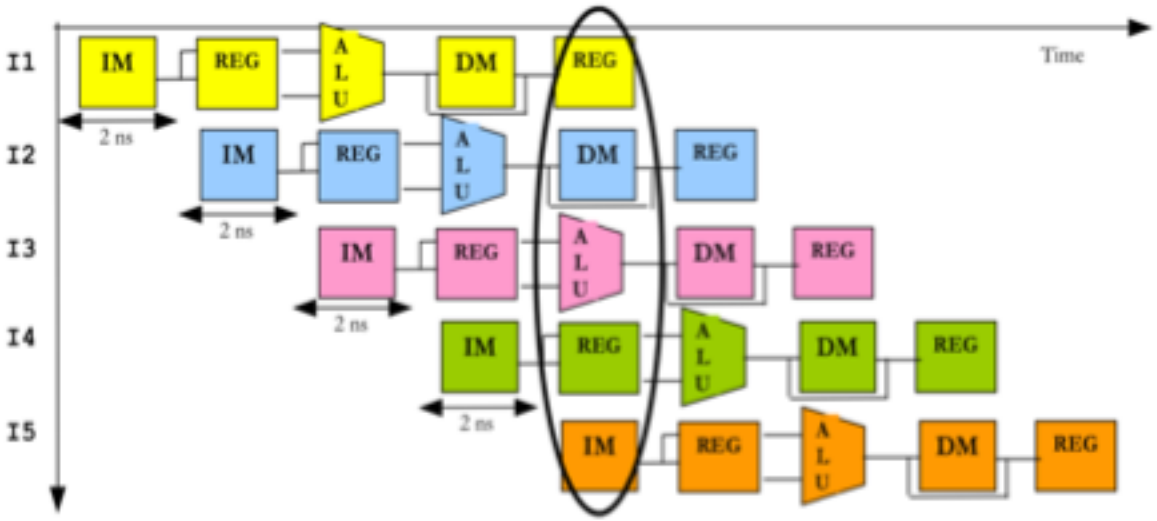


Single Cycle vs Multi Cycle vs Pipeline

	Single Cycle	Multiple Cycle	Pipeline
Instructions Subdivided?	No	Yes, into arbitrary number of steps	Yes, into one step per pipeline stage
Clock Cycle Time	Long (long enough for the slowest instruction)	Short (long enough for the slowest instruction step)	Short (long enough for the slowest pipeline stage)
CPI	1 clock cycle per instruction (by definition)	Variable number of clock cycles per instruction	Fixed number of clock cycles per instruction, one for each pipeline stage (under ideal conditions)
Number of Instructions Executing at the Same Time	1	1	As many instructions as pipeline stages (under ideal conditions)
Control Unit	Generates signals for entire instruction	Generates signals for instruction's current step, and keeps track of the current step	Generates signals for entire instruction; these signals are propagated from one pipeline stage to another via the pipeline registers
Duplicate Hardware?	Yes, since we can use a functional unit (FU) for at most one subtask per instruction	No, since the instruction generally is broken into single-FU steps	Yes, so that there are no restrictions on which instructions can be in the pipeline simultaneously
Extra Registers?	No	Yes, to hold the results of one step for use in the next step	Yes, to provide the results of one pipeline stage to the next pipeline stage
Performance	Baseline	Slightly slower to moderately faster than single cycle, the latter when the instructions steps are well balanced and a significant fraction of the instructions take less than the maximum number of clock cycles	Moderately faster to significantly faster than single cycle, the latter if the pipeline stages are well balanced and the pipeline handles hazards well

Pipeline: Hazards

- **Hazard (conflict)** is created whenever there is a dependence between instructions, and instructions are close enough that the overlap caused by pipelining would change the order of access to the operands involved in the dependence.
- Hazards prevent the next instruction in the pipeline from executing during its designated clock cycle.
- Hazards reduce the performance from the ideal speedup gained by pipelining

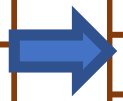


- **Structural Hazards**: Attempt to use the same resource from different instructions simultaneously
  - Example: Single memory for instructions and data
- **Data Hazards**: Attempt to use a result before it is ready
  - Example: Dependency between instructions
- **Control Hazards**: Attempt to make a decision on the next instruction to execute before the condition is evaluated
  - Example: Conditional branch execution

Pipeline: Structural Hazard

- In pipelining technique, the overlapped execution of instructions requires pipelined functional units or duplication of resources to allow all possible combinations of instructions. If the resource is conflicted by the overlapped stages of instructions → **Structural Hazard**
- Common instances of structural hazards arise when
  - Some functional unit is not fully pipelined. Then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle.
  - Some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute.
- Examples:
  - CPU has only 1 write port into RF, but in some cases the pipeline might want to perform two writes in a clock cycle → **Solution: allow 2 write port into RF**
  - CPU has unique memory for data and instructions → **Solution: insert Bubbles or Stall**

Clock cycle number								
Instr	1	2	3	4	5	6	7	8
Load	IF	ID	EX	MEM	WB			
Instr 1		IF	ID	EX	MEM	WB		
Instr 2			IF	ID	EX	MEM	WB	
Instr 3				IF	ID	EX	MEM	WB



Clock cycle number									
Instr	1	2	3	4	5	6	7	8	9
Load	IF	ID	EX	MEM	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Instr 3				stall	IF	ID	EX	MEM	WB

Pipeline: Data Hazard

- **Data hazards** arises if the instruction executed in the pipeline **are dependent and are too close**
- Consider the pipelined execution of these instructions: **SUB**, **AND** and **OR**
- Data Hazard classification:
  - *RAW (READ AFTER WRITE) hazard*
    - Ex: ADD R1, R2, R3 SUB R4, R1, R5
  - *WAW (WRITE AFTER WRITE) hazard*
    - Ex: LD R1, R2 (2 MEM stages) ADD R1, R2, R3 (without MEM stage)
    - Ex: MUL F6, F2, F2 (4 EX/MUL stages) ADD F6, F2, F2 (2 EX/ADD stage)
  - *WAR (WRITE AFTER READ) hazard*
    - Ex: LD R1, (R2) ADDI R2, R2, 4

		1	2	3	4	5	6	7	8	9
ADD	**R1**, R2, R3	IF	ID	EX	MEM	**WB**				
SUB	R4, R5, **R1**		IF	**ID**$_{sub}$	EX	MEM	WB			
AND	R6, **R1**, R7			IF	**ID**$_{and}$	EX	MEM	WB		
OR	R8, **R1**, R9				IF	**ID**$_{or}$	EX	MEM	WB	
XOR	R10, **R1**, R11					IF	**ID**$_{xor}$	EX	MEM	WB

Pipeline: Data Hazard

- **Compilation Techniques:**
  - Insertion of ***NOP*** *(no operation)* instructions
  - Instructions scheduling to avoid that correlating instructions are too close
    - The compiler tries to insert independent instructions among correlating instructions
    - When the compiler doesn't find independent instructions, it insert **NOPS**

- **Hardware Techniques:**
  - **Insertion of *stalls* or *"bubbles"* in the pipeline**
  - **Data forwarding or bypassing**

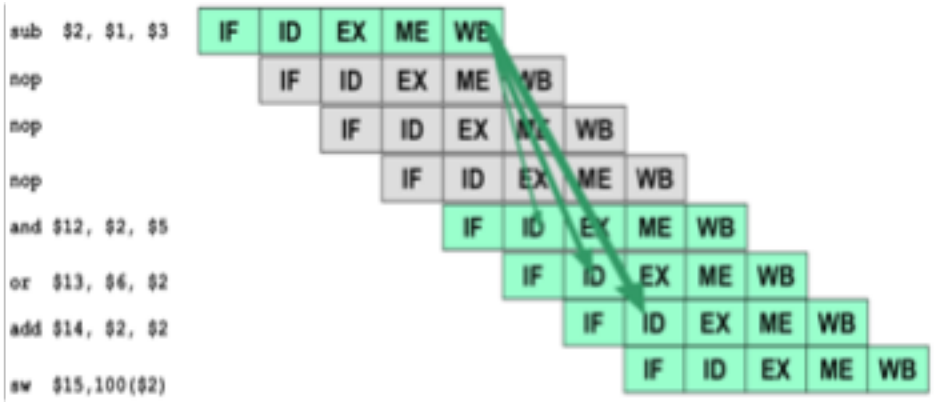
Insert stalls

Insert independent instructions

```
sub  $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15,100($2)
add $4, $10, $11
and $7, $8, $9
lw $16, 100($18)
```

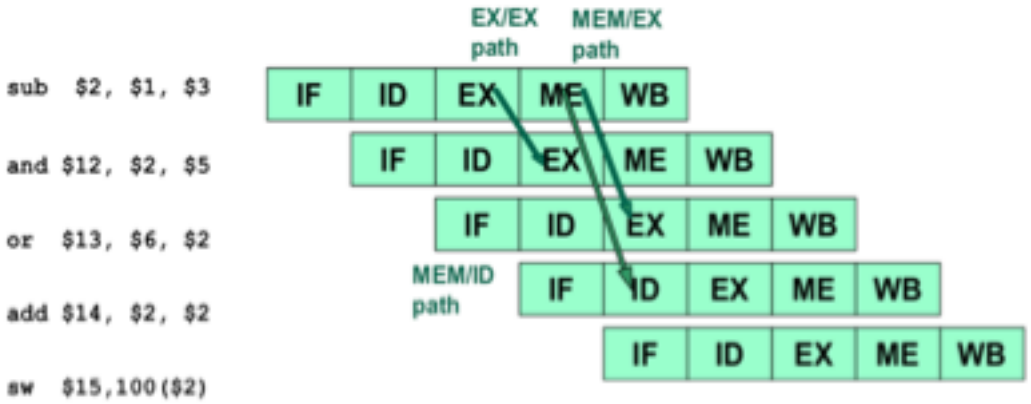
```
sub  $2, $1, $3
add $4, $10, $11
and $7, $8, $9
lw $16, 100($18)
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15,100($2)
```

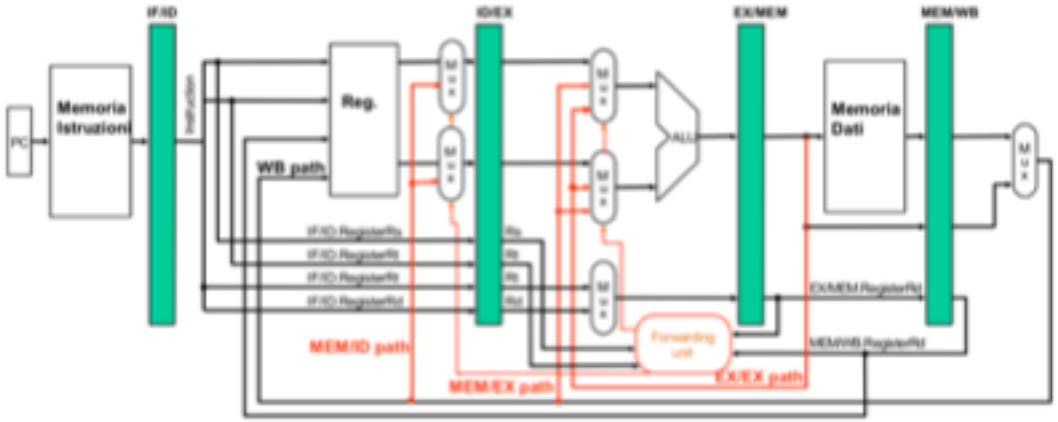
Insert NOP



Pipeline: Data Hazards

- Data forwarding uses *temporary results* stored in the pipeline registers instead of waiting for the write back of results in the RF
- We need to add multiplexers at the inputs of ALU to fetch inputs from pipeline registers to avoid the insertion of stalls in the pipeline





Pipeline: Control Hazard

- ***Control hazards*** can cause a greater performance loss for pipeline than data hazards. When a branch is executed, it may or may not change the PC. If a branch changes the PC to its target address, it is a ***taken branch***; if it falls through, it is ***not taken***
  - If instruction i is not a taken branch, then the PC will be changed in IF
  - If instruction i is a taken branch, then the PC will be changed at the end of MEM
- To deal with branch, ***Stall*** is essential to be inserted as soon as the branch is detected
- Stall in control hazard MUST be implemented differently from data hazard
  - **IF** cycle of the instruction following the branch **must be** essentially a ***stall*** (because it never performs useful work), which comes to total 3 stalls → significant lost → could be a ***half*** the speedup of ideal pipelining
- The number of clock cycles can be reduced by:
  - Find out whether the branch is taken or not taken ***earlier*** in the pipeline
  - Compute the taken PC (i.e., the address of the branch target) ***earlier***

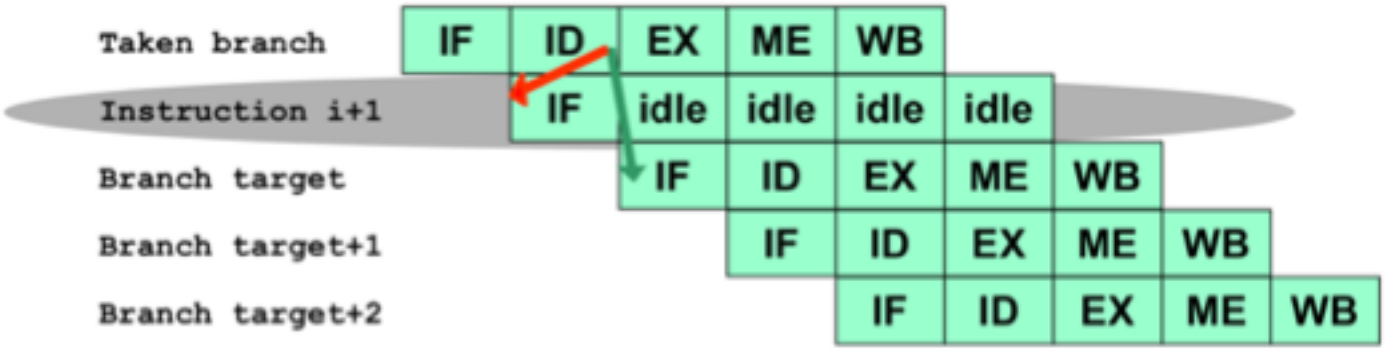
Branch	IF	**ID**	EX	**MEM**	WB					
Branch successor		**IF**(stall)	***stall***	***stall***	IF	ID	EX	MEM	WB	
Branch successor+1						IF	ID	EX	MEM	WB

Pipeline: Branch Prediction

- Main goal of branch prediction techniques: try to predict asap the outcome of a branch instruction to reduce to cost (number of cycles).
- The performance of branch prediction techniques depend on:
  - *Branch frequency* measured in terms of number of branch instructions appear in a program.
  - *Accuracy* measured in terms of percentage of incorrect prediction given by predictor.
  - *Cost of incorrect prediction* measured in terms of time lost to execute useless instruction (misprediction penalty) give by the processor of architecture.
- **Static Branch Prediction**: the action for a branch is fixed (at compile time) during the the entire execution
  - Branch Always Not Taken (Predicted-Not-Taken)
  - Branch Always Taken (Predicted-Taken)
  - Backward Taken Forward Not Taken (BTFNT)
  - Profile-Driven Prediction
  - Delayed Branch
- **Dynamic Branch Prediction**: the decision of a branch can dynamically change during the program execution: *1 bit History Branch Table*, *2 bit History Branch Table*, N bit History Branch Table

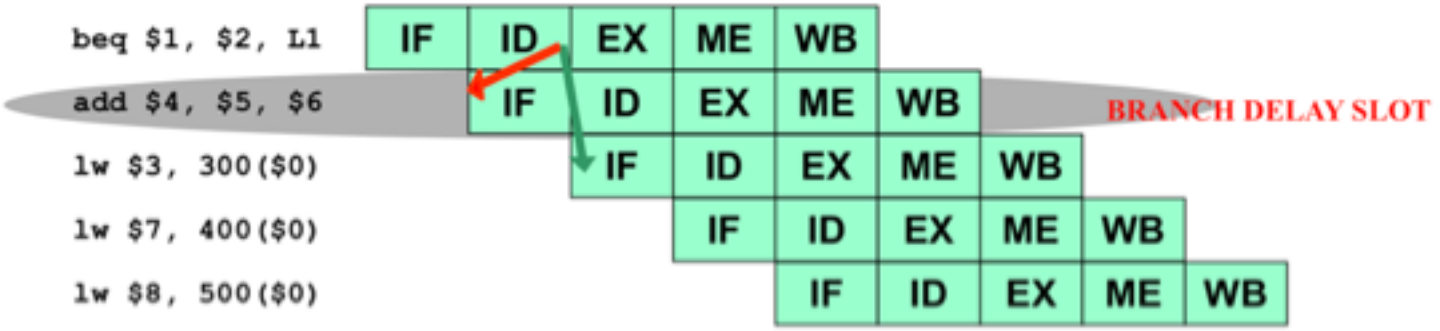
Pipeline: Static Branch Prediction

- For **Branch Predict Not Taken**, we assume the *branch will not be taken*
  - If the condition in stage ID will result not satisfied (the prediction is correct), *the next instruction already fetched can continue* → preserve the performance
  - If the condition in stage ID will result satisfied (the prediction is incorrect), *the branch is taken*:
    - Next instruction, which is already fetched, will be flushed (the fetched instruction is turned into a **nop**)
    - Restart the execution by fetching the branch target → 1 cycle is lost (incorrect prediction penalty)
- If 50% of branches are not taken → the cost of control hazards is reduce a half. The probability of Not taken is high, the cost could be optimized



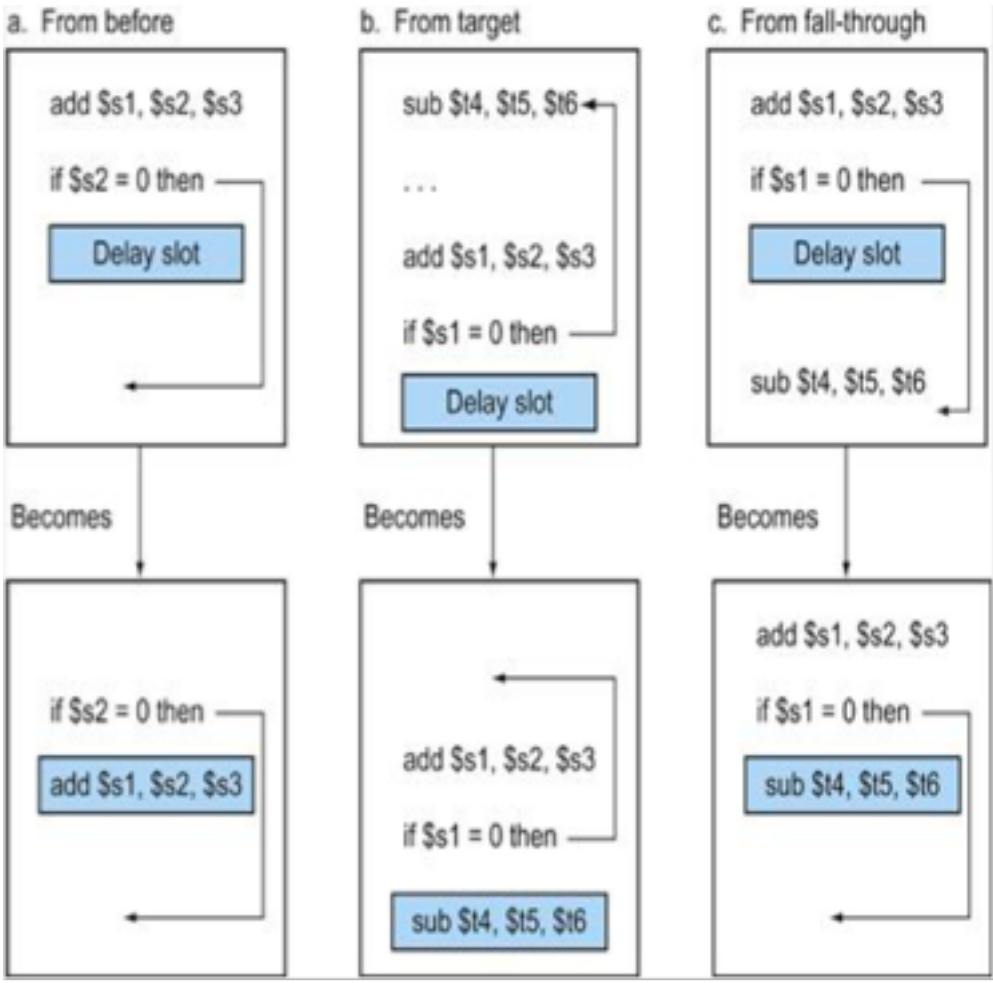
Pipeline: Static Branch Prediction

- For **Delay Branch technique**, the job of the compiler is to schedule an independent instruction in the branch delay slot valid and useful. The instruction in the branch delay slot is executed whether or not the branch is taken.
- There are four ways in which the branch delay slot can be scheduled:
  - From before
  - From target
  - From fall-through
  - From after
- Example: A previous **ADD** instruction with any effects on the branch is scheduled in the **Branch Delay Slot**



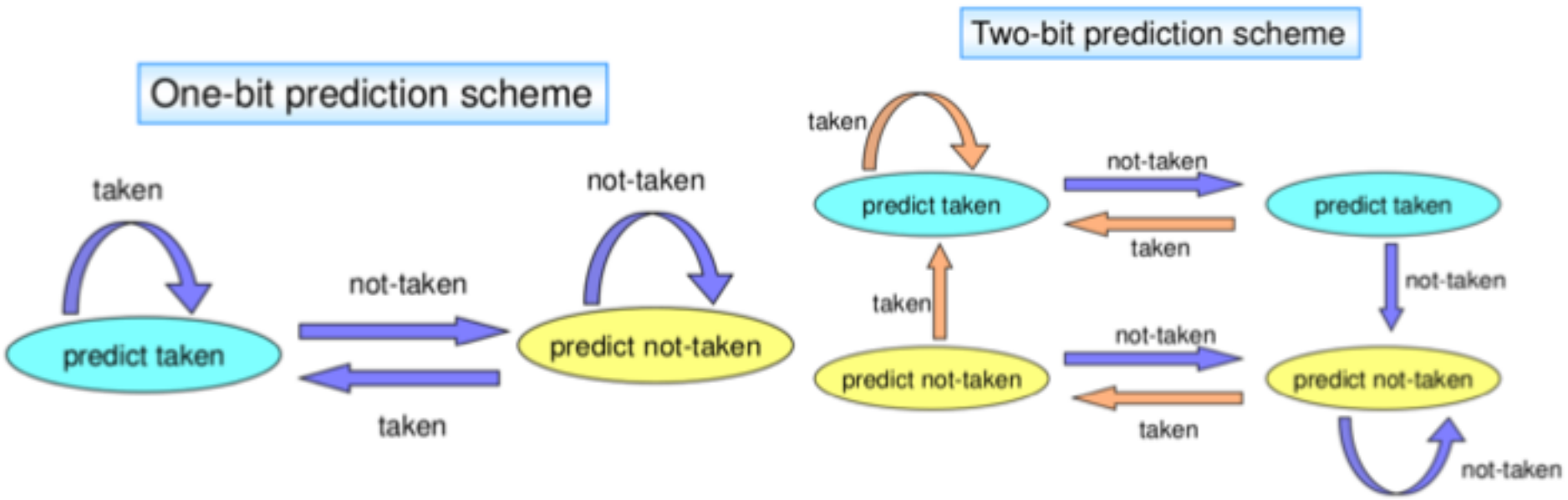
Pipeline: Static Branch Prediction

- **From before technique**: The branch delay slot is scheduled with an independent instruction from before the branch. The instruction in the branch delay slot is **always executed** (whether the branch is taken or untaken)
- **From target technique**
  - The use of **$1** in the branch condition prevents **add** instruction (whose destination is **$1**) from being moved after the branch.
  - The branch delay slot is scheduled from the target of the branch (usually the target instruction will need to be copied because it can be reached by another path).
  - This strategy is preferred when the branch is taken with high probability, such as loop branches **(backward branches)**
- **From fall through**
  - The use of $1 in the branch condition prevents add instruction (whose destination is $1) from being moved after the branch.
  - The branch delay slot is scheduled from the not-taken fall-through path.
  - This strategy is preferred when the branch is not taken with high probability, such as **forward branches**



Pipeline: Dynamic Branch Prediction

- Basic Idea: to use the past branch behavior to predict the future.
- Method: Using hardware to dynamically predict the outcome of a branch. The prediction will depend on the behavior of the branch at run time and will change if the branch changes its behavior during execution.



Pipeline: Super pipelining

- Superpipeline is the technique developed to improve the performance beyond what mere pipeline can offer.
- Superpipeline improves the performance by decomposing the long latency stages (such as memory access stages) of a pipeline into several shorter stages, thereby possibly increasing the number of instructions running in parallel at each cycle → Super-pipelining seeks to improve the *sequential* instruction rate by reducing the clock cycle time (fast clock)
- For example, by dividing each stage into two sub-stages, a pipeline can perform at twice the speed in the ideal situation.
  - Many pipeline stages may perform tasks that require less than half a clock cycle.
  - No duplication of hardware is needed for these stages
- For a given architecture and the corresponding instruction set there is an optimal number of pipeline stages/sub-stages.

Pipeline: Super pipelining

- Increasing the number of stages/sub-stages over this limit reduces the overall performance.
  - Overhead of data buffering between the stages.
  - Not all stages can be divided into (equal-length) sub stages.
  - The hazards will be more difficult to resolved.
  - The clock skew problem.
  - More complex hardware.
- **MIPS R4000** is an early version of superpipeline architecture with 8 stages compared to MIPS R3000. More over, MIPS R4000 operates at 200 MHz meanwhile MIPS R3000 operates at 100 MHz.
- **The ARM11** is a high- performance and low power processor which is equipped with 8 stages pipelining.
- **The modern processors are based on a combination of superpipelining and superscalar architecture**, i.e. The Intel NetBurst Microarchitecture with hyperpipeline technology, high clock rates (up to 10 GHz), different parts of processor can run at different clock rates → Intel Pentium 4, Intel Xeon with 31 pipelined stage

Year	Microprocessor	Clock Rate MHz	Pipeline Depth
1978	8086	4.77	2
1982	80286	8	3
1985	80386	16	3
1989	80486	25	5
1993	Pentium	66	5
1995	Pentium Pro	150	12
1997	Pentium 2	233	12
1999	Pentium 3	450	12
2000	P4 Willamette	1400	20
2001	Itanium	733	10
2002	Itanium 2	1000	8
2003	Pentium M	1730	10
2004	P4 Prescott	3400	31
2005	Pentium D	2800	31
2006	Core 2 Conroe	2930	14
2008	Core 2 Yorkfield	2930	16
2010	Core i7 Gulftown	3460	16

Question?