License: https://creativecommons.org/licenses/by-nc-nd/4.0/

2. The Pipeline

In **pipelining**, **m**ultiple tasks (for example, instructions) are executed in parallel.

To use the pipelining approach efficiently

1. We must have tasks that are repeated many times on different data.
2. Tasks must be divided into small pieces (operations or actions) that can be performed in parallel.

Example of a pipeline: an automobile assembly line.

The task

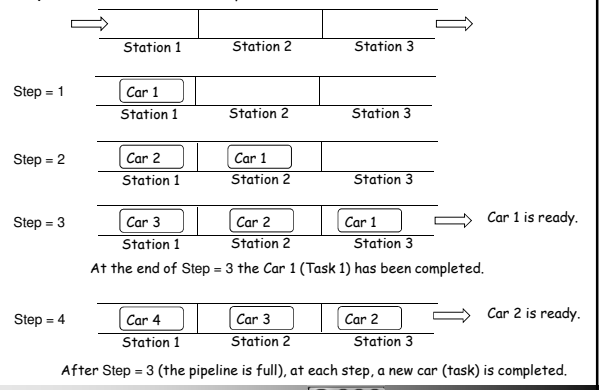
- is the construction of a car,
- is repeated many times for different cars,
- consists of some operations, such as attaching the doors, attaching the tires.

Each operation

- has its own station in the pipeline (assembly line).
- is performed in parallel with other operations but on a different car.

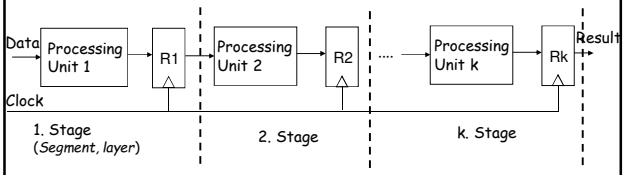
  e.g., while a worker is attaching the doors of the i^{th} car, another worker is attaching the tires of the $(i+1)^{st}$ car at the same time.

Example: An automobile assembly line with three stations



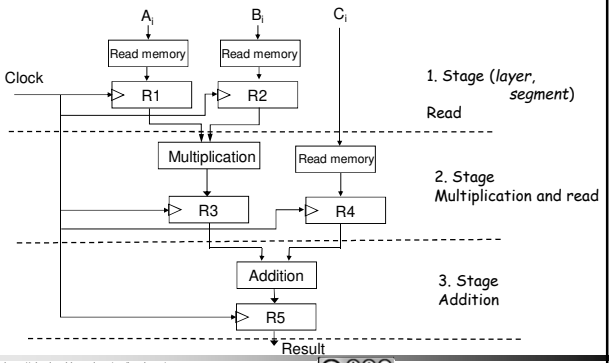
After Step = 3 (the pipeline is full), at each step, a new car (task) is completed.

2.1 The general structure of a pipeline:



- Each processing unit performs a fixed operation.
- In each clock cycle, the operation is performed on different data (task). (*Refer to Digital Circuits Lecture notes, Section 6 for information about clock signal.*)
- Registers (R1, R2, …, Rk) keep the intermediate results.
- All stages are controlled by a common clock signal and operate synchronously.
- New inputs are accepted at one end, before previously accepted inputs appear as outputs at the other end.
- When all stages of the pipeline are full, in each clock cycle, a new result is produced at the output.

Example: The elements of the arrays A, B, and C will be first read from memory, and then the following operation will be performed: $\mathbf{A_i * B_i + C_i}$ i=1,2,3,...



Example (cont'd):

- In this example, the task is decomposed into 3 operations: Reading, multiplication, and addition.
- We assume that arrays are in separate memory modules, which can be read in parallel.
- We start to read elements of array C one clock cycle after reading A and B.

Functioning of the pipeline with three stages:

Clock cycle	1. Stage (Read) R1	R2	2. Stage(Multiply) R3	R4	3.Stage (Add) R5
1	A_1	B_1	-	-	-
2	A_2	B_2	A_1*B_1	C_1	-
3	A_3	B_3	A_2*B_2	C_2	$A_1*B_1 + C_1$ (First result)
4	A_4	B_4	A_3*B_3	C_3	$A_2*B_2 + C_2$ (2nd result)
5	A_5	B_5	A_4*B_4	C_4	$A_3*B_3 + C_3$ (3rd result)

Note:

- Assuming that the time to access the memory is significantly shorter than the durations of the other operations and the data is always ready to be read, reading is not treated as a separate operation.
- In this case, the pipeline could be designed with two stages which perform only arithmetical operations: multiplication and addition.

2.2 Space-Time Diagram of a pipeline with four stages

Space-time diagrams (or timing diagrams) show which task is currently being processed in which stage of the pipeline.

In the exemplary diagram below, clock cycles (steps) are the column labels, stages are the row labels (Si) , and task numbers (Ti) are the table entries.

Example:
(4 stages)

Time → Clock Cycles (steps)

Stages	1	2	3	4	5	6	7
S1	T1	T2	T3	T4	T5	T6	
S2		T1	T2	T3	T4	T5	T6
S3			T1	T2	T3	T4	T5
S4				T1	T2	T3	T4

The 1st task (T1) is completed in 4 clock cycles (number of stages k=4).

After the k^{th} cycle, a new task is completed in each clock cycle.

Four tasks (T4) have been completed in 7 clock cycles.

License: https://creativecommons.org/licenses/by-nc-nd/4.0/

Space-Time Diagram of a pipeline with four stages, cont'd

We could also construct the space-time diagram in an alternative way.

In the diagram below, clock cycles (steps) are the column labels, tasks (Ti) are the row labels, and stages (Si) are the table entries.

Time ⟶ Clock Cycles (steps)

Tasks	1	2	3	4	5	6	7
T1	S1	S2	S3	S4			
T2		S1	S2	S3	S4		
T3			S1	S2	S3	S4	
T4				S1	S2	S3	S4

The 1st task (T1) is completed in 4 clock cycles (number of stages k=4)

After the k^{th} cycle, a new task is completed in each clock cycle

Four tasks (T4) have been completed in 7 clock cycles.

2.3 Throughput and Speedup provided by the pipeline

Since all stages proceed at the same time, the time (delay) required for the **slowest stage** determines the length of the period of the clock signal (cycle time).

The cycle time (the period of the clock) t_p can be determined as follows:

$$t_p = \max(\tau_i) + d_r = \tau_M + d_r$$

t_p: cycle time

τ_i : time delay of the circuitry in the i^{th} stage

τ_M : maximum stage delay (the slowest stage)

d_r : time delay of the register

Speedup:

k: number of stages in the pipeline
t_p: cycle time
n: number of tasks
t_n : time required for a task without pipelining

Calculation of the total time required for n tasks:

- k cycles required to complete the first task (T1). Time: $T(1) = k \cdot t_p$
- remaining n-1 tasks require (n-1) cycles. + Time: $(n-1)t_p$

$\Rightarrow$ Total time required for n tasks: $T(n) = (k+n-1)tp$

Speedup: $S = \dfrac{\textit{Execution time } \textbf{\textit{without}} \textit{ the pipeline}}{\textit{Execution time } \textbf{\textit{with}} \textit{ the pipeline}}$ $\quad S = \dfrac{n \cdot t_n}{(k+n-1) \cdot t_p}$

If the number of tasks increases significantly : $n \to \infty$, $\lim_{n\to\infty} S = \dfrac{t_n}{t_p}$

If we assume $t_n = k \cdot t_p$,
(If it were possible to divide the main task into k equal small operations and ignore the register delays, the cycle time would be $t_p = t_n / k$.)

S_{max} = k (**Theoretical** maximum speedup)

Comments on speedup:

To improve the performance of the pipeline, tasks must be divided into small and balanced operations with equal (or at least similar) durations.

If the durations of the operations are short, then the clock cycle (t_p) can be short.

Remember: The slowest stage determines the clock cycle.

Effects of increasing the number of stages of a pipeline:

Advantage:

- If the task can be divided into **many small** operations, increasing the number of stages can lower the clock cycle (t_p), and consequently the speedup increases.

$\lim_{n\to\infty} S = \dfrac{t_n}{t_p}$ $\quad S_{max} = k$ (Theoretical)

Disadvantages:

- The cost of the pipeline increases. At each stage of the pipeline, there is some overhead (cost, energy, space) because of registers and additional connections.
- The completion time of the first task increases. $T(1) = k \cdot t_p$
- Branch penalties in the instruction pipeline caused by control hazards increase. We will discuss branch penalties in the section "2.5 Pipeline hazards".

While designing a pipeline, these advantages and disadvantages should be taken into consideration.

Effects of task partitioning on the speedup:

If the task can be partitioned into small operations with small durations then a faster clock signal (shorter cycle time) can be used.

Assume that we have a task T with a total duration of 100 ns.

Assume that we can decompose this task in different ways.

Case A: We partition the task into 2 equal stages.

T:	S1 = 50ns	S2 = 50ns

If the delay of the registers is 5 ns, then the clock cycle is $t_p = 50+5 = 55$ ns

Case B: We partition the task into 3 unbalanced stages.

T:	S1 = 25ns	S2 = 25ns	S3 = 50ns

The clock cycle is $t_p = 50+5 = 55$ ns (slowest stage τ_M =50ns)

Although the pipeline has more stages, there is no speed improvement compared to case A, because t_p is still 55 ns .

Besides, the cost of the pipeline has increased.

Also, the completion time of the first task has increased. $T(1) = k \cdot t_p$

Effects of task partitioning on the speedup: (cont'd)

Case C: We partition the task into three stages with similar durations.

T:	S1 = 30ns	S2 = 30ns	S3 = 40ns

The clock cycle is $t_p = 40+5 = 45$ ns (slowest stage τ_M = 40ns)

The clock rate ($1 / t_p$) is higher compared to cases A and B.

Conclusion:

Pipelining has advantages if a task can be partitioned into small and balanced operations.

It is important to decrease the length of the clock cycle (t_p).

For example, if we could partition the task into five operations, each having the duration of 20ns, we would have a clock cycle of 25ns.

License: https://creativecommons.org/licenses/by-nc-nd/4.0/

2.4 Instruction Pipeline (Instruction-Level Parallelism)

During the execution of each instruction the CPU repeats some operations.

The processing required for a single instruction is called an **instruction cycle**.

An instruction cycle is generally composed of these stages: instruction fetch and decoding, operand fetch, execution, interrupt. (See the figure on 1.18)

The simplest instruction pipeline can be constructed with two stages:

1) Fetch and decode instruction 2) Fetch operands and execute instruction

When the main memory is not being accessed during the execution of an instruction, this time can be used to fetch the next instruction in parallel with the execution of the current one.

Example:

Cycle:	1	2	3	4
Instr. 1	Fetch, decode	Operand, exec.		
Instr. 2		Fetch, decode	Operand, exec.	
Instr. 3			Fetch, decode	Operand, exec.

The potential overlap among instructions is called **instruction-level** parallelism.

Remember: To gain more speedup, the pipeline must have more stages with short durations.

Instruction Pipeline (cont'd)

The instruction cycle can be decomposed into 6 operations to gain more speedup:

1. **Fetch instruction (FI):** Read the next expected instruction into a buffer.
2. **Decode instruction (DI):** Determine the opcode and the operand specifiers.
3. **Calculate addresses of operands (CO):** Calculate the effective address.
4. **Fetch operands (FO):** Fetch each operand from memory.
5. **Execute instruction (EI):** Perform the indicated operation.
6. **Write operand (WO):** Store the result in memory.

Such fine-grained decomposition may not significantly increase the performance because of the following problems :

- The various stages will be of different durations (unbalanced).
- Some instructions do not need all stages.
- Different segments may need the same resources (e.g., memory) at the same time.

Therefore, some operations can be combined into the same stage so that a pipeline with fewer (for example 4 or 5), balanced stages is constructed.
For example, the 80486 had 5 stages.

There are also processors that include instruction pipelines with more stages.
For example, Pentium 4 family processors have a pipeline with 20 stages. In these processors, internal operations are decomposed into microoperations.

2.4.1 An (exemplary) instruction pipeline (with 4 stages)

1. FI (*Fetch Instruction*): Read the next instruction the PC points to into a buffer.
2. DA (*Decode, Address*): Decode instruction, calculate operand addresses
3. FO (*Fetch Operand*): Read operands (memory/register)
4. EX (*Execution*): Perform the operation and update the registers (including the PC in branch/jump instructions)

- In order to perform instruction and operand fetch operations at the same time, we assume that the processor has separate instruction and data memories.
- Memory-write operations are ignored in these examples.
- This an exemplary pipelined CPU. More realistic examples are given in section "2.4.2 An Exemplary RISC Processor with Pipelining".

2.4.1 An (exemplary) instruction pipeline (cont'd)

A) Ideal Case: No branches, no operand dependencies in the program

Timing diagram for the exemplary instruction pipeline (ideal case):

Clock cycles / Instructions (Tasks)	1	2	3	4	5	6	7
1	FI	DA	FO	EX			
2		FI	DA	FO	EX		
3			FI	DA	FO	EX	
4				FI	DA	FO	EX

The first instruction has been completed. 4 cycles
The pipeline is full.

After just one cycle, the second instruction has been completed.

The first instruction was completed in 4 cycles (k=4).

After the 4th cycle, a new instruction is completed in each cycle.

If the number of instructions approaches infinity, the completion time of an instruction approaches 1 cycle (slide 2.9 "Speedup").

2.4.1 An (exemplary) instruction pipeline (cont'd)

B) Pipeline Hazards (Conflicts)

B.1 Data Conflict (Operand dependency):

The operand of an instruction depends on the result of another instruction

Example :

Clock cycles / Instructions	1	2	3	4	5
ADD R1, **R2** (**R2** ← R1+R2)	FI	DA	FO	EX	
SUB **R2**, R3 (R3 ↔ **R2**-R3)		FI	DA	FO	EX

R2 is updated.

Operand dependency

Previous value (not valid) of R2 is being fetched.

To prevent the program from running incorrectly, a solution mechanism must be applied.

For example: The pipeline can be stopped (stall), or NOOP (No Operation) instructions can be inserted.

We will discuss possible solutions in the section "2.5 Pipeline Hazards (Conflicts) and Solutions".

2.4.1 An (exemplary) instruction pipeline (cont'd)

B.2 Control Hazards (Branches, Interrupts):

Since a pipeline processes instructions in parallel, during the processing of a branch instruction, the next instruction in the memory that should be actually skipped also enters the pipeline.

Here, a solution mechanism is necessary; otherwise, the instruction(s) that should be skipped according to the program will also be executed.

Example:

```
1.          Instruction_1
2.          JUMP Target
3.          Instruction_3
              :
4. Target   Instruction_4
```

Unconditional branch (or jump) instruction (BRA / JUMP)

Next instruction in the memory According to the program, it **should be skipped**.

Target of the branch (target instruction)

During the processing of the unconditional branch instruction JUMP, Instruction_3 is also fetched into the pipeline.

To prevent the program from running incorrectly, the pipeline must be stopped (stall) or emptied before Instruction_3 is executed.

License: https://creativecommons.org/licenses/by-nc-nd/4.0/

a. Unconditional Branch

Clock cycles / Instructions	1	2	3	4	5	6	7
Instruction 1	FI	DA	FO	EX			
Instruction 2 **JUMP**		FI	DA	FO	EX		
Instruction 3			~~FI~~	-	-		
Target Instr. 4						FI	DA

Steps

After decoding, the type of the instruction is determined: branch!

The branch address is fetched (absolute or relative).

Updating the PC (program counter) PC = Target (Target of branch)

Hazard: This instruction is fetched unnecessarily. It must not be executed. It will (must) be discarded.

Branch penalty! It is necessary to *stall* or *empty* the pipeline.

The new instruction after branch operation (Target of branch)

After decoding (identification) of the unconditional branch instruction, one possible solution is to stop the "Fetch Instruction" stage (FI) of the pipeline.

After the execution of the branch instruction, the target address is written to the program counter (PC), and the pipeline is enabled to fetch new instructions.

b. Conditional Branch:

For a conditional branch instruction, there are two cases:

1. condition is false (branch is not taken), 2. condition is true (branch is taken)

b1. Conditional Branch (if the condition is false):

If the condition is not true, it is not necessary to stop or empty the pipeline because the execution will continue with the next instruction.

Clock cycles / Instructions	1	2	3	4	5	6
Instruction 1	FI	DA	FO	EX		
Conditional bra. 2		FI	DA	FO	EX	
Instruction 3			FI	DA	FO	EX

The previous instruction sets the conditions (flags).

PC is not changed. No branching.

The instruction following the branch is executed.

Without considering the condition, next instruction is fetched.

No need to empty
No branch penalty

Here, the problem is that the previous instruction must be executed to determine if the condition is true or not (depends on the flags of the CPU).

- If condition is false (branch is not taken), there is no branch penalty.
- If condition is true, a solution mechanism is necessary (next slide).

b2. Conditional Branch (if the condition is true):

Clock cycles / Instructions	1	2	3	4	5	6	7
1	FI	DA	FO	EX			
Conditional bra. 2		FI	DA	FO	EX		
3			~~FI~~	DA	FO		
4				~~DA~~	DA		
5					~~FI~~		
Target 6						FI	DA

Condition is true.
The branch address is written to PC.
PC = Target
The pipeline must be emptied.

The pipeline is emptied.

Branch penalty: 3 clock cycles

The target instruction of branch

The duration of the branch penalty depends on the number and the operations of the stages in the pipeline.
In this exemplary pipeline, the branch penalty is 3 clock cycles; however, it may be different in other types of pipelines (2.5.3. Control Hazards).

2.4.2 An Exemplary RISC Processor with Pipelining

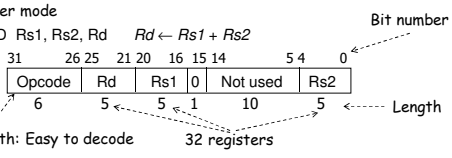
- Instructions are fixed-length (commonly 32 bits).
  This simplifies fetch and decode operations (advantage in pipelining).
- Most instructions are register-to-register. Only for load and store operations memory-to-register and register-to-memory instructions are necessary.
- There are few addressing modes.
- Some exemplary instructions:
  - ADD Rs1,Rs2,Rd — *Rd ← Rs1 + Rs2*
    ADD R3, R4, R12 — *R12 ← R3 + R4*
  - ADD Rs,S2,Rd — *Rd ← Rs + S2* (S2: immediate data)
    ADD R1, #$1A, R2 — *R2 ← R1 + $1A*
  - LDL S2(Rs),Rd — *Rd←M[Rs + S2]* Load long (32 bits)
    LDL $500(R4), R5 — *R5 ← M[R4 + $500]*
  - STL S2(Rs), Rm — *M[Rs + S2] ← Rm* Store long (32 bits)
    STL $504(R6), R7 — *M[R6 + $504] ← R7*
  - BRU Y — *PC←PC + Y* Unconditional branch
    BRU $0A — *PC←PC + $0A* Branch relative (Y: Offset)
  - Bcc Y — *If (cc) then PC←PC + Y* Conditional branch
    BGT $0A — *If greater, then PC←PC + $0A*

Instruction Formats of the Exemplary RISC Processor

- Three different instruction types:

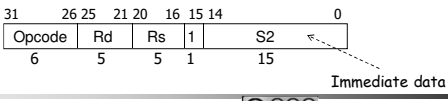
1. Register mode

ADD Rs1, Rs2, Rd *Rd ← Rs1 + Rs2*



2. Immediate mode

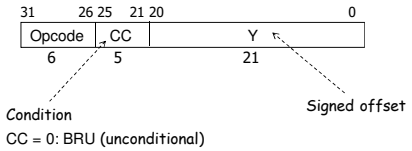
- ADD Rs, S2, Rd *Rd ← Rs + S2* (S2: immediate data)
- LDL S2(Rs), Rd *Rd←M[Rs + S2]* Load long (32 bits)



Instruction Formats of the Exemplary RISC Processor (cont'd)

3. Relative

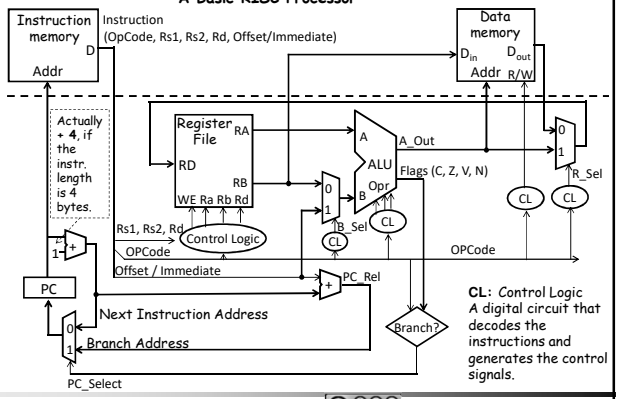
- BRU Y *PC←PC + Y* Unconditional branch
- Bcc Y *If (cc) then PC←PC + Y* Conditional branch



CC = 0: BRU (unconditional)

License: https://creativecommons.org/licenses/by-nc-nd/4.0/

A Basic RISC Processor



Pipelined RISC Alternatives

There are different ways of designing pipelined RISC processors.
For example;

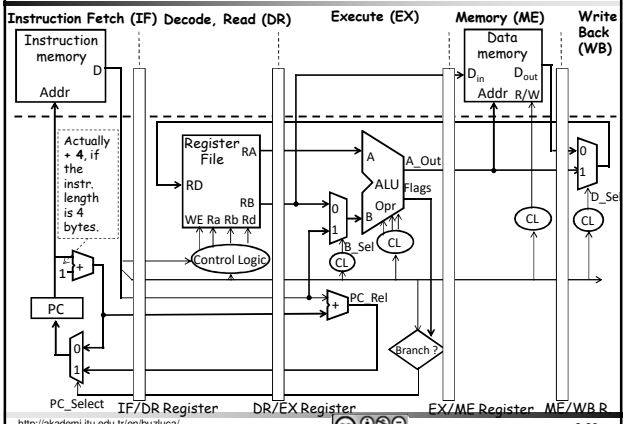
- ARM7 has 3 stages
  IF: Instruction fetch;
  DR: Decode and read registers;
  EX: ALU Operation; access memory (if necessary), write the result to the registers
- MIPS R3000 has 5 stages
- MIPS R4000 has 8 stages (superpipelined)
- ARM Cortex-A8 has 13 stages.

An Exemplary 5-Stage RISC Pipeline

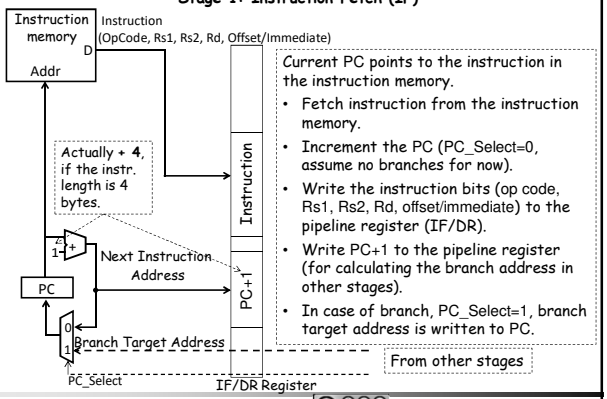
In this course, to explain the concepts, we will use an exemplary five-stage RISC load-store architecture :

1. Instruction fetch (IF):
   Get instruction from memory, increment PC (depending on the instruction length).
   If instruction length is 4 bytes, PC ← PC + 4.
2. Instruction Decode, Read registers (DR)
   Translate opcode into control signals and read registers (operands).
3. Execute (EX)
   Perform ALU operation, compute jump/branch targets.
4. Memory (ME)
   Access memory if needed (only load/store instructions).
5. Write back (WB)
   Update register file (write results).

A 5-Stage RISC Pipeline



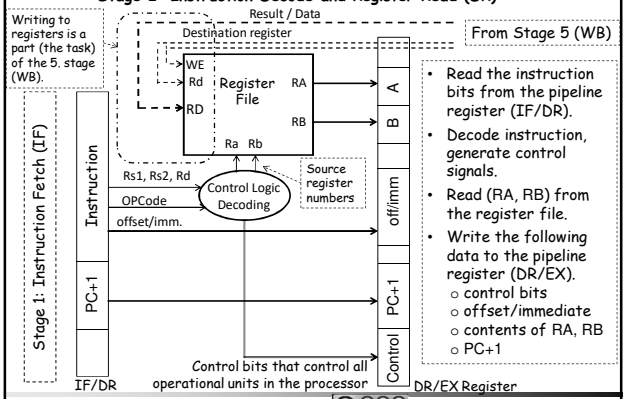
Stage 1: Instruction Fetch (IF)



Current PC points to the instruction in the instruction memory.

- Fetch instruction from the instruction memory.
- Increment the PC (PC_Select=0, assume no branches for now).
- Write the instruction bits (op code, Rs1, Rs2, Rd, offset/immediate) to the pipeline register (IF/DR).
- Write PC+1 to the pipeline register (for calculating the branch address in other stages).
- In case of branch, PC_Select=1, branch target address is written to PC.

Stage 2: Instruction Decode and Register Read (DR)



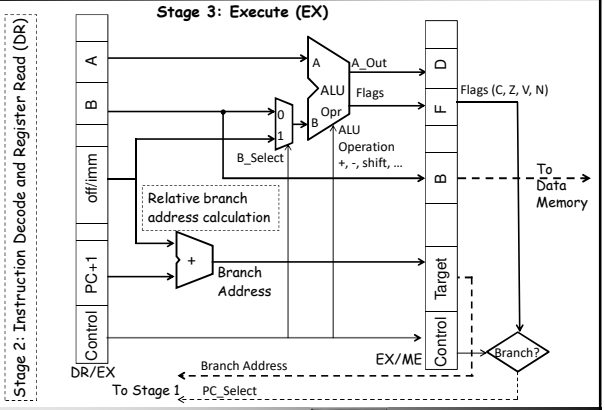
- Read the instruction bits from the pipeline register (IF/DR).
- Decode instruction, generate control signals.
- Read (RA, RB) from the register file.
- Write the following data to the pipeline register (DR/EX).
  - control bits
  - offset/immediate
  - contents of RA, RB
  - PC+1

License: https://creativecommons.org/licenses/by-nc-nd/4.0/

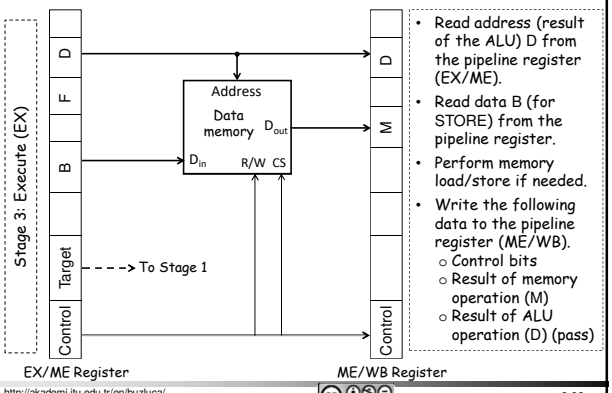
Stage 3: Execute (EX)

- Read the control bits and data (offset/immediate, RA, RB) from the pipeline register (DR/EX).
- Perform the ALU operation.
  The ALU also calculates memory addresses for LOAD/STORE instructions.
  For example; LDL $500(R4), R5 *R5 ← M[R4 + $500]*
  The immediate value $500 is added to the contents of R4 in the ALU.
- Compute target addresses for the branch instructions
  For example: BGT $0A *If greater, then PC←PC + $0A*
  In this exemplary processor, an additional adder is used for target address calculation.
- Decide if the jump/branch should be taken (control bits and flags from the ALU are used)
- Write the following data to the pipeline register (EX/ME):
  - Control bits
  - Result of the ALU (D) and flags (F)
  - RB for memory store operations (B)
  - Branch target address

Stage 3: Execute (EX)

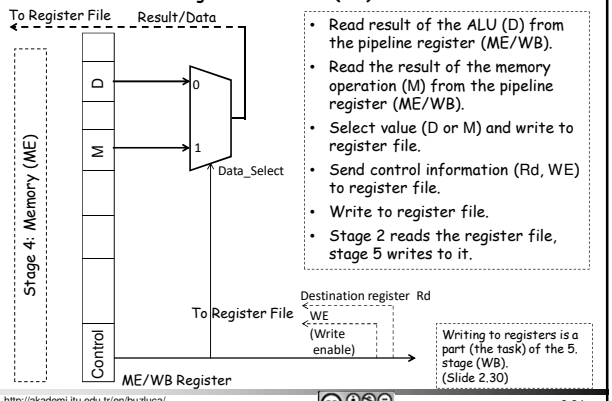


Stage 4: Memory (ME)



- Read address (result of the ALU) D from the pipeline register (EX/ME).
- Read data B (for STORE) from the pipeline register.
- Perform memory load/store if needed.
- Write the following data to the pipeline register (ME/WB).
  - Control bits
  - Result of memory operation (M)
  - Result of ALU operation (D) (pass)

Stage 5: Write Back (WB)



- Read result of the ALU (D) from the pipeline register (ME/WB).
- Read the result of the memory operation (M) from the pipeline register (ME/WB).
- Select value (D or M) and write to register file.
- Send control information (Rd, WE) to register file.
- Write to register file.
- Stage 2 reads the register file, stage 5 writes to it.

Timing diagram for the exemplary RISC pipeline (ideal case):

Ideal Case: No branches, no conflicts

Clock cycles / Instructions	1	2	3	4	5	6	7	8
1	IF	DR	EX	ME	WB			
2		IF	DR	EX	ME	WB		
3			IF	DR	EX	ME	WB	
4				IF	DR	EX	ME	WB

The first instruction has been completed. 5 cycles Pipeline is full.

Just after one cycle, the second instruction has been completed.

The first instruction was completed in 5 cycles (k = 5).

After the 5th cycle, a new instruction is completed in each cycle.

If the number of instructions approaches infinity, the completion time of an instruction approaches 1 cycle (see slide 2.9 "Speedup").

IF and ME stages try to access the memory at the same time.

To solve the resource conflict problem, separate memories for instruction and data are used (Harvard architecture).

2.5 Pipeline Hazards (Conflicts) and Solutions

There are three types of hazards

1. Resource Conflict (Structural hazard):

A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource (memory, functional unit).

2. Data Conflict (Hazard)

Data hazards occur when data is used before it is ready.

3. Control Hazards (Branch, Jump, Interrupt):

During the processing of a branch instruction, the next instruction in the memory that should actually be skipped also enters the pipeline.

Which **target instruction** should be fetched into the pipeline is unknown, unless the CPU executes the branch instruction (updating the PC).

Conditional branch problem: Until the actual execution of the instruction that alters the flag values, the flag values are unknown (greater?, equal?), so it is impossible to determine if the branch will be taken.

Stalling solves all these conflicts but it reduces the performance of the system.

There are more efficient solutions.

License: https://creativecommons.org/licenses/by-nc-nd/4.0/

2.5.1. Resource Conflict (Structural hazard):

A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource (memory, functional unit).

a) Memory conflict: "An operand read to or write from memory cannot be performed in parallel with an instruction fetch."

Solutions:

- Instructions must be executed serially rather than in parallel for a portion of the pipeline (*stall*). (Performance drops.)
- Harvard architecture: Separate memories for instructions and data.
- Instruction queue or cache memory: There are times during the execution of an instruction when main memory is not being accessed. This time could be used to prefetch the next instruction and write it to a queue (instruction buffer).

b) Functional unit (ALU, FPU) conflict.

Solutions:

- Increasing available functional units and using multiple ALUs.

For example, different ALUs can be used address calculation and data operations.

- Fully pipelining a functional unit (for example, a floating point unit FPU)

2.5.2. Data Conflict (Hazard):

Data hazards occur when data is used before it is ready.

If the problem is not solved, the program may produce an incorrect result because of the use of pipelining.

Example:

ADD R1, R2, **R3** $R3 \leftarrow R1 + R2$

SUB **R3**, R4, R5 $R5 \leftarrow R3 - R4$

Data dependency in the pipeline

Clock cycles / Instructions	1	2	3	4	5	6
ADD R1,R2,**R3**	IF	DR	EX	ME	WB	
SUB **R3**,R4,R5		IF	DR	EX	ME	WB

Result of ADD is written to the register file (R3).

SUB reads R3 before it has been updated. R3 does not contain the result of the previous ADD instruction; it has not been processed in WB yet.

2.5.2. Data Conflict (cont'd):

There are three types of data hazards:

- **Read after write (RAW), or true dependency:** An instruction modifies a register or memory location, and a succeeding instruction reads the data in that memory or register location.

  A hazard occurs if the read takes place before the write operation is complete.

- **Write after read (WAR), or antidependency:** An instruction reads a register or memory location, and a succeeding instruction writes to the location.

  A hazard occurs if the write operation completes before the read operation takes place.

- **Write after write (WAW), or output dependency:** Two instructions both write to the same location.

  A hazard occurs if the write operations take place in the reverse order of the intended sequence.

Solutions to Data Hazards:

A) Stalling, Hardware interlock (Hardware-based solution):

An additional hardware unit tracks all instructions (control bits) in the pipeline registers and stops (stalls) the instruction fetch (IF) stage of the pipeline when a hazard is detected.

The instruction that causes the hazard is delayed (not fetched) until the conflict is solved.

Example:

Clock cycles / Instructions	1	2	3	4	5	6	7	8	9
ADD R1,R2,**R3**	IF	DR	EX	ME	WB				
SUB **R3**,R4,R5		IF	-	-	-	DR	EX	ME	WB

First write to R3, then read it. Write and read in different clock cycles.

Data conflict is detected. IF/DR.Rs1 == DR/EX.Rd

Pipeline is stalled. 3-clock-cycles delay

Stalling the pipeline:

- IF/DR register is disabled (no update).
- Control bits of the NOOP (*No Operation*) instruction are inserted into the DR stage.
- The PC is not updated.

Solutions to Data Hazards (cont'd):

Fixing the register file access hazard:

The register file can be accessed in the same cycle for reading and writing.

Data can be written in the first half of the cycle (rising edge) and read in the second half (falling edge).

This method reduces the waiting (stalling) time from 3 cycles to 2 cycles.

Clock cycles / Instructions	1	2	3	4	5	6	7	8
ADD R1,R2,**R3**	IF	DR	EX	ME	WB			
SUB **R3**,R4,R5		IF	-	-	DR	EX	ME	WB

First write to R3 in the first half, then read it in the second half.

Data conflict is detected. IF/DR.Rs1 == DR/EX.Rd

Write

Read

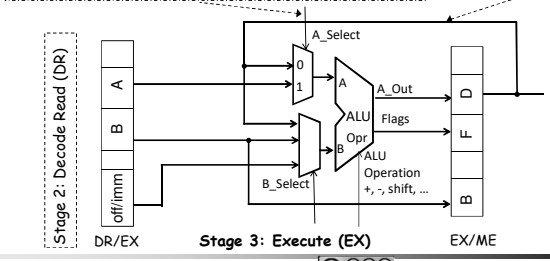
Solutions to Data Hazards (cont'd):

B) Operand forwarding (Bypassing) (Hardware-based):

An optional direct connection is established between the output of the EX stage (EX/ME register) and the inputs of the ALU.

A_Select and B_Select are controlled by the hazard detection unit of the pipeline. It selects either the value from the register file or the forwarded result (bypass) as the ALU input.

Forwarding (Bypass)



License: https://creativecommons.org/licenses/by-nc-nd/4.0/

Operand forwarding (Bypassing) from EX/ME to ALU (cont'd**):**

If the hazard unit detects that the destination of the previous ALU operation is the same register as the source of the current ALU operation, the control logic selects the forwarded result (bypass) as the ALU input, rather than the value from the register.

Example:

Clock cycles / Instructions	1	2	3	4	5
ADD R1, R2, **R3**; R3←R1 + R2	IF	DR	EX	ME	WB
SUB **R3**, R4, R5; R5←R3 - R4		IF	DR	EX	ME

Previous value (not valid) of R3 is fetched.
This invalid value will **not be used** in the EX cycle.

The control unit of the pipeline selects the output of the previous ALU operation (*bypass*) as the input, not the value that has been read in the DR stage (A_Select = 0).

If it is possible to solve the register conflict by forwarding, it is not necessary to stall the pipeline.

The performance does not drop.

Solution to load-use data hazard using Operand forwarding (Bypassing)

Load-use data hazard:

Load instructions may also cause data hazards.

Example:

Data from memory is written to the register file (R1).

Load-use data hazard

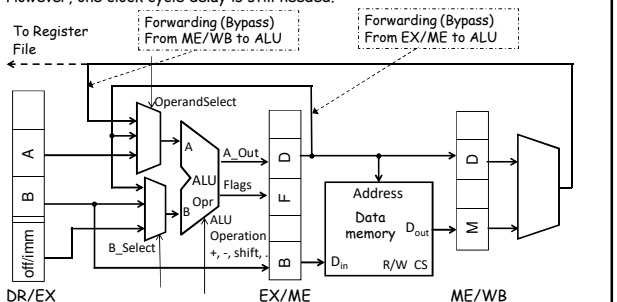
Clock cycles / Instructions	1	2	3	4	5	6
LDL $500(R4), **R1** *R1 ← M[R4 + $500]*	IF	DR	EX	ME	WB	
ADD **R1**, R2, R3 *R3 ← R1 + R2*		IF	DR	EX	ME	WB

ADD reads R1 before it has been updated.
The value in R1 is not valid.

Operand forwarding (Bypassing) from ME/WB to ALU:

To decrease the waiting time caused by load-use hazard, an optional direct connection can be established between the output of the ME stage (ME/WB register) and the inputs of the ALU.

However, one clock cycle delay is still needed.



Load_use data hazard (cont'd):

Solution with forwarding + 1 cycle stalling

Example:

Solution with forwarding (+stalling)

Clock cycles / Instructions	1	2	3	4	5	6	7
LDL $500(R4), R1	IF	DR	EX	ME	WB		
ADD R1, R2, R3		IF	-	DR	EX	ME	WB

The previous value (not valid) of R1 is fetched.
This invalid value will **not be used** in the EX cycle.

The control unit of the pipeline selects the forwarding path as the input, not the value that has been read in the DR stage.

Solutions to Data Hazards (cont'd):

C) Inserting NOOP (No Operation) instructions (Software-based):

The effect of this solution is similar to stalling the pipeline.

The **compiler** inserts NOOP instructions between the instructions that cause the data hazard.

Example:

Clock cycles / Instructions	1	2	3	4	5	6	7	8
ADD R1,R2,**R3**	IF	DR	EX	ME	WB			
Inserted by the compiler **NOOP**		IF	DR	EX	ME	WB		
NOOP			IF	DR	EX	ME	WB	
SUB **R3**,R4,R5				IF	DR	EX	ME	WB

First write to R3 in the first half, then read it in the second half.

Since NOOP is a machine language instruction of the processor, it is processed in the pipeline just like other instructions.

The performance drops because of the delay caused by the NOOP instructions.

Solutions to Data Hazards (cont'd):

D) Optimized Solution (Software-based):

The compiler rearranges the program and moves certain instructions (if possible) between the instructions that cause the data hazard.

This rearrangement must not change the algorithm or cause new conflicts.

Example:

STL $00(R6), R1 *M[R6 + $00] ← R1*
STL $04(R6), R2 *M[R6 + $04] ← R2*
ADD R1, R2, **R3** *R3 ← R1 + R2*
SUB **R3**, R4, R5 *R5 ← R3 – R4*

Write to R3 in the first half, read it in the second half.

Clock cycles / Instructions	1	2	3	4	5	6	7	8
ADD R1,R2,**R3**	IF	DR	EX	ME	WB			
Moved by the compiler **STL $00(R6), R1**		IF	DR	EX	ME	WB		
STL $04(R6), R2			IF	DR	EX	ME	WB	
SUB **R3**,R4,R5				IF	DR	EX	ME	WB

The performance is improved.

There is no delay caused by NOOP instructions (or stalling).

License: https://creativecommons.org/licenses/by-nc-nd/4.0/

2.5.3. Control Hazards (Branches, Interrupts):

In the exemplary RISC processor, the following operations are performed for the branch/jump instructions:

- The target address for the branch (jump) instruction is calculated in the Execution (EX) stage (slide 2.32).
- The target address is written to the EX/ME pipeline register.
- The branch decision is made in the Memory (ME) stage based on the values of flags that are determined after the execution in the EX stage (slide 2.32).
- After the EX stage, the result of the decision (PC_Select) and the target address are sent to Stage 1 (IF).
- In the IF stage, the next instruction the PC points to is fetched first, then the PC is updated (slide 2.29).

During these operations, the next instructions in sequence (not the target of branch) are fetched into the pipeline.

However, if the branch is taken, these instructions should be skipped.

In this case, either a hardware unit must empty the pipeline, or compiler-based solutions (delayed branch) must be applied.

The unnecessary instructions must be stopped before they are processed in the WB stage because the registers of the CPU are changed in that stage.

Conditional Branch Hazards:

Example:

```
100 SUB   R1, R2, R1     R1 ← R1 - R2
104 BGT   $1C            Branch if greater ($108 + $1C = $124 Target address)
108 ADD   R1, R1, R2
10C ADD   R3, R4, R2
110 STL   $00(R5), R2
114 LDL   $0A(R6), R1
...
124 STL   $00(R6), R2    Target of BGT
```

These instructions should be skipped if the branch is taken.

Remember: Bcc conditional branch instructions check the flag values generated by the last ALU operation.

For example, the BGT instruction (signed comparison) checks the flags N (Negative), V (Overflow), and Z (Zero).

After the operation **R = A – B** the table on the right is used to compare the signed integers.

Overflow (V)	Sign (N)	Zero (Z)	Comparison
X (not important)	Positive (0)	YES (1)	A=B
NO (0)	Positive (0)	NO (0)	A>B
NO (0)	Negative (1)	NO (0)	A<B
YES (1)	Positive (0)	NO (0)	A<B
YES (1)	Negative (1)	NO (0)	A>B

Conditional Branch Hazards (cont'd):

Example (cont'd): **If the branch is taken**

The target address ($108 + $1C = $124) has been calculated in EX and written to the EX/ME register.

The branch decision is made (After EX). "Take the branch"

The target address is sent from the EX/ME register to IF stage.

Instructions										
SUB R1, R2, R1	IF	DR	EX	ME	WB					
BGT $1C		IF	DR	EX	ME	WB				
ADD R1, R1, R2			IF	DR	EX	ME	WB			
ADD R3, R4, R2				IF	DR	EX	ME	WB		
STL $00(R5), R2					IF	DR	EX	ME	WB	
Target: STL $00(R6), R2						IF	DR	EX	ME	WB

These instructions should be skipped.

The pipeline must be stalled (emptied by hardware), or a compiler-based solution must be applied.

The PC is updated at the end of the IF. PC← $124 (Target)

The target instruction of BGT is fetched.

In the case of a stall, the **branch penalty is 3 cycles** for this exemplary CPU.

Conditional Branch Hazards (cont'd):

Example (cont'd): **If the branch is NOT taken**

The target address is sent from the EX/ME register to IF stage. This address is not used, because branch is NOT taken.

The target address ($108 + $1C = $124) has been calculated in EX and written to the EX/ME register.

The branch decision is made (After EX). "NO branch"

Instructions										
SUB R1, R2, R1	IF	DR	EX	ME	WB					
BGT $1C		IF	DR	EX	ME	WB				
ADD R1, R1, R2			IF	DR	EX	ME	WB			
ADD R3, R4, R2				IF	DR	EX	ME	WB		
STL $00(R5), R2					IF	DR	EX	ME	WB	
LDL $0A(R6), R1						IF	DR	EX	ME	WB

The PC is updated at the end of the IF. PC← PC+1 (Next instruction) **Not the target address of the branch.**

Next instruction in sequence.

If the branch is not taken, there is no branch penalty.

Reducing the branch penalty:

Conditional branch:

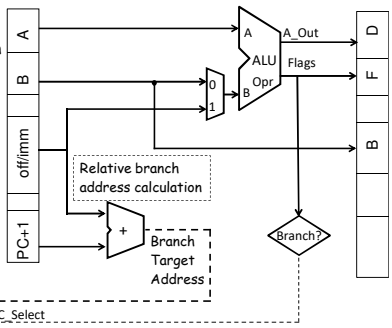
The Execution (EX) stage is modified.

Branch target address calculation and decision operations are performed in the EX stage, and results are sent directly to the IF stage.

In the case of a stall, we will have **2 cycles** (instead of 3) of branch penalty, if the branch is taken (slide 2.54).

The decision operation will increase the delay of the EX stage.

Execute (EX) stage



Reducing the branch penalty (cont'd):

Conditional branch (cont'd) : If branch is taken

Example:

The target address ($108 + $1C = $124) has been calculated. The branch decision has been made (In EX).

The target address is sent to the IF stage.

Instructions										
SUB R1, R2, R1	IF	DR	EX	ME	WB					
BGT $1C		IF	DR	EX	ME	WB				
ADD R1, R1, R2			IF	DR	EX	ME	WB			
ADD R3, R4, R2				IF	DR	EX	ME	WB		
Target: STL $00(R6), R2					IF	DR	EX	ME	WB	

These instructions should be skipped.

The pipeline must be stalled (emptied by hardware), or a compiler-based solution must be applied.

The PC is updated at the end of the IF. PC← $124 (Target)

The target instruction of BGT is fetched.

In the case of a stall, the **branch penalty is 2 cycles** for this exemplary pipeline.

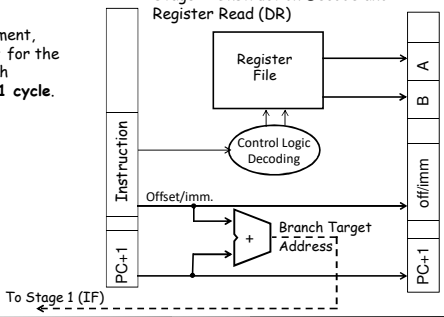
License: https://creativecommons.org/licenses/by-nc-nd/4.0/

Reducing the branch penalty (cont'd):

Unconditional branch:

Because the flag values are not needed, the branch target address calculation can be moved into the DR stage.

After this improvement, the **branch penalty** for the unconditional branch instruction BRU is **1 cycle**.



Reducing the branch penalty (cont'd):

Unconditional branch (cont'd) :

Example:

The target address ($108 + $1C = $124) has been calculated.

The target address is sent to the IF stage.

Instructions								
SUB R1, R2, R1	IF	DR	EX	ME	WB			
BRU $1C		IF	DR	EX	ME	WB		
Should be skipped. ADD R1, R1, R2			IF	~~DR~~	~~EX~~	~~ME~~	~~WB~~	
Target: STL $00(R6), R2				IF	DR	EX	ME	WB

The pipeline must be stalled (emptied by hardware), or a compiler-based solution must be applied.

The PC is updated at the end of the IF. PC← $124 (Target)

The target instruction of BRU is fetched.

For the unconditional branch instruction, the **branch penalty is 1 cycle** after moving the address calculation operation to the DR stage.

Solutions to Control (Branch) Hazards:

A) Stalling/flushing (hardware-based):

A hardware unit detects the hazards and stalls the pipeline until the target instruction is fetched.

Stalling can be applied to both unconditional and conditional branch hazards.

Example: Unconditional branch, target address calculation is in DR

BRU (hazard) is detected.

The target address is calculated.

The target address is sent to the IF stage.

Instructions								
SUB R1, R2, R1	IF	DR	EX	ME	WB			
BRU $1C		IF	DR	EX	ME	WB		
ADD R1, R1, R2			IF	-	-	-	-	
Target: STL $00(R6), R2				IF	DR	EX	ME	WB

This instruction is removed from the pipeline.

The PC is updated at the end of the IF. PC← $124 (Target)

The target instruction of BRU is fetched.

Solutions to Control (Branch) Hazards (cont'd):

B) Inserting NOOP (No Operation) instructions (Software-based):

The **compiler** inserts NOOP instructions after the branch instruction.

The effect of this solution is similar to stalling the pipeline.

Example: Unconditional branch, address calculation is in DR stage

The target address has been calculated.

The target address is sent to the IF stage.

Instructions								
SUB R1, R2, R1	IF	DR	EX	ME	WB			
BRU $1C		IF	DR	EX	ME	WB		
Inserted by the compiler. **NOOP**			IF	DR	EX	ME	WB	
Target: STL $00(R6), R2				IF	DR	EX	ME	WB

The PC is updated at the end of the IF. PC ← Target

The target instruction of BRU is fetched.

B) Inserting NOOP (No Operation) instructions (cont'd):

The number of NOOP instructions that need to be inserted depend on the number of stall cycles required.

Example: Conditional branch;
address calculation and branch decisions are in EX (slide 2.53).

In this case, 2 stall cycles are necessary. Therefore, 2 NOOPs are inserted.

The target address ($108 + $1C = $124) has been calculated.
The branch decision has been made (In EX).

The target address is sent to the IF stage.

Instructions									
SUB R1, R2, R1	IF	DR	EX	ME	WB				
BGT $1C		IF	DR	EX	ME	WB			
Inserted by the compiler. **NOOP**			IF	DR	EX	ME	WB		
NOOP				IF	DR	EX	ME	WB	
Target: STL $00(R6), R2					IF	DR	EX	ME	WB

The PC is updated at the end of the IF. PC ← Target

The target instruction of BGT is fetched.

Solutions to Control (Branch) Hazards (cont'd):

C) Optimized Solution (Software-based):

The compiler rearranges the program and moves certain instructions (if possible) to immediately after the branch instruction.

This rearrangement must not change the algorithm or cause new conflicts.

Example: Unconditional branch, address calculation is in DR stage

```
SUB   R1, R2, R1
BRU   $1C
ADD   R3, R4, R2
STL   $00(R6), R2
```

The target address has been calculated.

The target address is sent to the IF stage.

Instructions								
BRU $1C	IF	DR	EX	ME	WB			
Moved by the compiler. **SUB R1, R2, R1**		IF	DR	EX	ME	WB		
Target: STL $00(R6), R2			IF	DR	EX	ME	WB	

This instruction is fetched before the branch instruction updates the PC. The program is not changed.

The PC is updated at the end of the IF. PC ← Target

The target instruction of BRU is fetched.

If the optimized solution is possible, there is **no branch penalty**.

License: https://creativecommons.org/licenses/by-nc-nd/4.0/

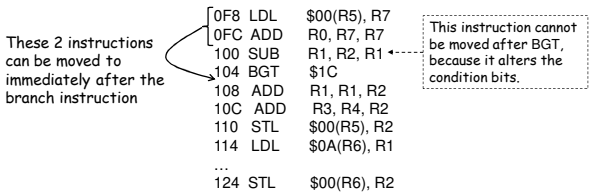
C) Optimized Solution (cont'd):

The number of instructions to be moved depends on the number of necessary stall cycles.

This rearrangement must not change the algorithm or cause new conflicts.

Example: Conditional branch, address calculation and branch decisions are in EX.

In this case 2 stall cycles are necessary. Therefore, 2 instructions must be moved after the branch instruction.



Important points about changing the order of the instructions:

- An instruction **from before** the branch can be placed immediately after the branch.
  - The branch (condition or address) must not depend on the moved instruction.
  - This method (if possible) always improves the performance (compared to inserting NOOP).
  - Especially for **conditional branches**, this procedure must be applied carefully.

  If the condition that is tested for the branch is altered by the immediately preceding instruction, then the complier cannot move this instruction to after the branch.

Other possibilities:

The compiler can select instructions to move

- **From branch target**

- Must be OK to execute moved instruction even if the branch is not taken
- Improves performance when branch is taken

- **From fall through (else)**

- Must be OK to execute moved instruction even if the branch is taken
- Improves performance when branch is not taken

Solutions to Control (Branch) Hazards (cont'd):

D) Branch Prediction:

Remember: The existence of branch/jump instructions in the program causes two main problems:

1. The CPU does **not know the target instruction** to fetch into the pipeline until it calculates the target address of the branch instruction.

   PC ← PC + offset

   Later stages of the pipeline (not IF stage) carry out this calculation. Options:

   a) If address calculation is in EX and result is sent from EX/ME register to IF stage (slide 2.32), branch penalty = 3 cycles.

   b) If address calculation is in EX and result is directly sent to IF stage (slide 2.53), branch penalty = 2 cycles.

   c) If address calculation is in DR and result is directly sent to IF stage (slide 2.55), branch penalty = 1 cycle (valid for unconditional branch/jump instructions).

   To solve this problem, a **branch target table** (slide 2.66) is used to determine the target address in advance.

   The branch target table is a cache memory in the IF stage that keeps the addresses of the branch instructions and their target addresses.

Branch/jump instructions in the program cause two main problems (cont'd):

2. **Conditional branch** problem: Until the previous instruction is actually executed, it is impossible to determine whether the branch will be taken or not because the values of the flags are not known.

   If the branch is not taken, PC ← PC + 4 (1 instruction = 4 bytes for the exemplary RISC processor)

   If the branch is taken, PC ← PC + offset

   a) If the branch decision logic is in ME stage (after EX) (slide 2.32), branch penalty = 3 cycles.

   b) If the branch decision logic is in EX (slide 2.53), branch penalty = 2 cycles.

   To solve this problem, **prediction mechanisms** are used.

   When a conditional branch is recognized, a branch prediction mechanism predicts whether the branch will be taken or not.

   According to the prediction, either the next instruction in the memory or the target instruction of the branch is prefetched.

   If the prediction is correct, there is no branch penalty.

   In case of misprediction, the pipeline must be stopped and emptied.

D) Branch Prediction (cont'd):

There are two types of branch prediction mechanisms: **static** and **dynamic**.

Static branch prediction strategies:

a) Always predict not taken: Always assumes that the branch will not be taken and fetches the next instruction in sequence.

b) Always predict taken: Always predicts that the branch will be taken and fetches the target instruction of the branch.

   To determine the target of the branch in advance (without calculation), the **branch target table** is used (slide 2.66).

Studies analyzing program behavior have shown that conditional branches are taken more than 50% of the time.

Therefore, always prefetching from the branch target address should give better performance than always prefetching from the sequential path.

Branch target table (BTT): Target Instruction prefetch

"Always predict taken" strategy: Always fetches target instruction of the branch.

However, the CPU does **not know the target instruction** to fetch into the pipeline until it calculates the target address of the branch instruction.

To determine the target of the branch in advance, the **branch target table (BTT)** is used.

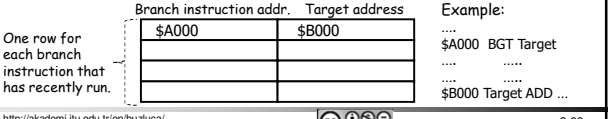
In the **branch target table**, addresses of recent branch instructions and their target addresses (where they jump) are kept in a cache memory (*Chapter 6*).

The BTT makes it possible for the target instruction to be prefetched in the 1. stage (IF) without calculating the branch target address.

There is a separate row for each branch instruction that has recently run.

The number of recent branch instructions stored is limited by the size of the table.

When a branch instruction runs for the first time, its target address is calculated and written into the BTT.



License: https://creativecommons.org/licenses/by-nc-nd/4.0/

D) Branch Prediction (cont'd):

Dynamic branch prediction strategies:

Dynamic branch strategies record the history of all conditional branch instructions in the active program to predict whether the condition will be true or not.

One or more **prediction bits** (or counters) are associated with each conditional branch instruction in a program that reflect the recent history of the instruction.

These prediction bits are kept in a **branch history table – BHT** (slide 2.69) and they provide information about the branch history of the instruction (branch was taken or not in previous runs).

1-bit dynamic prediction scheme:

For each conditional branch instruction (i), a single individual **prediction bit (p_i)** is stored in the branch history table (BHT).

The prediction bit $\mathbf{p_i}$ records only whether the last execution of this instruction (i) resulted in a branch or not.

If the branch was taken last time, the system predicts that the branch will be taken next time.

Algorithm:

Fetch the i^{th} conditional branch instruction

If ($p_i = 0$) then predict **not to take** the branch, fetch the next instruction in sequence

If ($p_i = 1$) then predict **to take** the branch, prefetch the target instruction of the branch

If the branch is really taken, then $p_i \leftarrow 1$

If the branch is not really taken, then $p_i \leftarrow 0$

The **initial value of p_i** is determined depending on the case in the first run of the conditional branch instruction.

In the first run, the target address is calculated and stored in the BHT.

During the calculation of the target address, next instructions in sequence (not the target of branch) are fetched into the pipeline. In case of a branch, there will be a branch penalty.

Branch target buffer and branch history table (BHT):

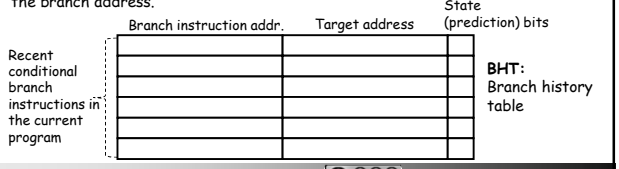
Prediction bits are kept in a high-speed memory location called the **branch history table (BHT)**.

For each recent branch instruction in the current program, the BHT stores the address of the instruction, the target address, and the state (prediction) bits.

Each time a conditional branch instruction is executed the associated prediction bits are updated according to whether the branch is taken or not.

These prediction bits direct the pipeline control unit to make the decision the next time the branch instruction is encountered.

If the prediction is that "the branch will be taken", with the help of the target buffer, the target instruction of the branch can be prefetched without calculating the branch address.



Example: 1-bit dynamic prediction scheme and loops:

Prediction mechanisms are advantageous if there are loops in the program.

Example:

```
        counter ← 100       ; register or memory location
LOOP    ----                ; instructions in the loop
        ----
        Decrement counter   ; counter ← counter - 1
        BNZ  LOOP           ; Branch if not zero (conditional branch, it has a p bit)
        ----                ; Next instruction after the loop
```

A) We assume that in the beginning of the given piece of code, the BNZ instruction is in the BHT and the value of its p bit is 1 (predict to take the branch).

In the first iteration (step) of the loop, the prediction at BNZ will be correct and the pipeline will prefetch the correct instruction (beginning of the loop).

The p bit (p=1) is not changed until the last iteration of the loop.

In the last iteration of the loop, the p bit is still 1, and the prediction is to take the branch; however, as the counter is zero, the program will not jump, and it will instead continue with the next instruction following the branch (misprediction).

The p bit of BNZ is cleared ($p \leftarrow 0$) because the branch is not taken in the last step.

As a result, in a loop with 100 iterations, there are 99 correct predictions and only one incorrect prediction.

Example: 1-bit dynamic prediction scheme and loops (cont'd):

B) If in the beginning of the given piece of code, the BNZ instruction is not in the BHT, the system cannot make a prediction in the first run.

After the calculation of the target address of the BNZ, the related information is written into the BHT.

During the calculation of the target address, next instructions in sequence (not the target of branch) are fetched into the pipeline.

In the first run, the branch is taken, and the program jumps to the beginning of the loop, so there will be a branch penalty.

The initial value of **p** becomes 1 (predict that the branch will be taken).

The value of p (p = 1) does not change until the last iteration (step) of the loop.

In the last iteration of the loop, the p bit is still 1, and the prediction is that the branch will be taken; however, as the counter is zero, the program will not jump, and it will instead continue with the next instruction following the branch (misprediction).

The p bit of BNZ is cleared ($p \leftarrow 0$).

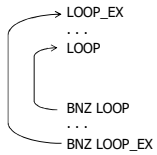
As a result, in a loop with 100 iterations, in the first iteration, a prediction cannot be made. Then, there are 98 correct predictions and one incorrect prediction. In total, there are 2 branch penalties.

Problem with the 1-bit dynamic prediction scheme:

(Nested loops: the same loop is executed many times)

In nested loops, a one-bit prediction scheme will cause **two** mispredictions for the inner loop:

- one in the first iteration, and
- one on exiting



Remember: in the previous example, after exiting the loop, the p bit of the inner BNZ LOOP was 0 ("don't take the branch") (p=0) .

Now, if the same loop runs again (2nd run), in the first iteration (step), the prediction about the BNZ will be "not to take the branch" (p=0).

However, the program will jump to the beginning of the loop (first misprediction).

Now, the p bit will be 1 because branch is taken ($p \leftarrow 1$).

Until the last iteration of the loop, predictions will be correct.

In the last iteration of the loop, there will be a misprediction as in the previous example (second misprediction).

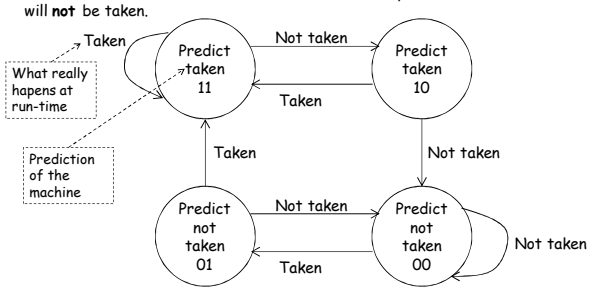
Hence, misprediction will occur **twice** for each full iteration of the inner loop.

License: https://creativecommons.org/licenses/by-nc-nd/4.0/

2-bit Branch prediction scheme:

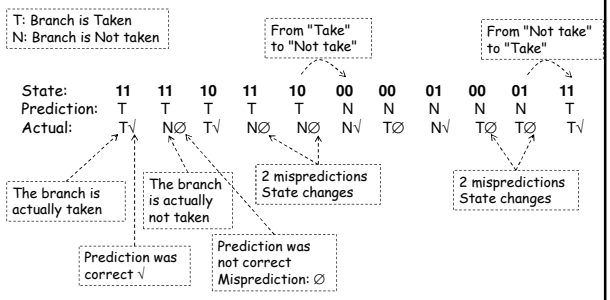
Two prediction bits are associated with each conditional branch instruction.

- If the instruction is in states 11 or 10, the scheme predicts that the branch will be taken.
- If the instruction is in states 00 or 01, the scheme predicts that the branch will **not** be taken.



In this scheme, the prediction changes only if it misses **twice**.

Example: 2-bit Branch prediction

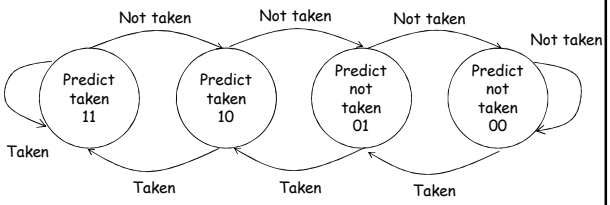


Saturating counter: Another 2-bit Branch prediction strategy

There are different ways of implementing the finite state machine for branch prediction strategies.

A Saturating counter is one of these alternatives.

- If the instruction is in states 11 or 10, the scheme predicts that the branch will be taken.
- If the instruction is in states 00 or 01, the scheme predicts that the branch will **not** be taken.



In this scheme, the prediction is changed only if it misses **twice** after one correct prediction.

Problem: **Example:**

A CPU has an instruction pipeline, where hardware-based mechanisms are used to solve branch hazards.
This CPU runs the given piece of code below, which includes two nested loops.

```
                Counter1 ← 10
-----> LOOP1    ------                      ; Any instruction
                Counter2 ← 10
->LOOP2         ------                      ; Any instruction
                ------                      ; Any instruction
                Counter2 ← Counter2 - 1
                BNZ   LOOP2                 ; Branch if not zero
                ------                      ; Instruction after loop2
                Counter1 ← Counter1 - 1
                BNZ   LOOP1                 ; Branch if not zero
                ------                      ; Instruction after loop1
```

For each branch prediction mechanism, give the number of correct predictions and mispredictions for the two branch instructions (BNZ) in the given piece of code.
Briefly explain your results.

Solution:

a. Static prediction

i) **Always predict not taken** (For this method, a BTT (branch target table) is not necessary)

BNZ LOOP1: There is a correct prediction only in the last iteration (exit).
Other predictions are incorrect.
Correct : 1 Incorrect : 9

BNZ LOOP2: There is a correct prediction only in the last iteration (exit).
Other predictions are incorrect.
Correct : 10x1 = 10 Incorrect : 10x9 = 90

Total: **Correct : 11** **Incorrect : 99**

This method is not suitable for loops.

a. Static prediction (cont'd)

ii-1) **Always predict taken** under the assumption that instructions are in the BTT

BNZ LOOP1: There is a misprediction only in the last iteration (exit).
Other predictions are correct.
Correct: 9 Incorrect: 1

BNZ LOOP2: There is a misprediction only in the last iteration (exit).
Other predictions are correct.
Correct : 10x9 = 90 Incorrect : 10x1 = 10

Total: **Correct: 99** **Incorrect: 11**

ii-2) **Always predict taken** under the assumption that instr. are NOT in the BTT

BNZ LOOP1: There are mispredictions only in the first and last iterations.
Other predictions are correct.
Correct: 8 Incorrect: 2

BNZ LOOP2: In the first run of the loop, there are mispredictions only in the first and last iterations; other predictions are correct.
In the 2nd -10th runs, there is a misprediction only in the last iteration (exit).
Correct : 8+9x9 = 89 Incorrect : 2+9x1 = 11

Total: **Correct: 97** **Incorrect: 13**

License: https://creativecommons.org/licenses/by-nc-nd/4.0/

Solution (cont'd):

b. Dynamic prediction with one bit

Note: Different prediction bits are used for each branch instruction (Slides 2.68, 2.69).

i) Assumption: In the beginning, instructions are in the BHT, and initial decision is to **take the branch**

BNZ LOOP1: There is a misprediction only in the last iteration (exit). Other predictions are correct.
Correct: 9 Incorrect: 1

BNZ LOOP2: In the first run of the loop, there is a misprediction only in the last iteration (exit).
Other predictions are correct.
After the first run, the prediction bit "p" changes to "branch will not be taken".
Therefore, in the 2nd-10th runs, there are mispredictions in both the first and last iterations (Slide 2.71).
Correct: 9 + 9x8 = 81 Incorrect: 1+ 9x2 =19

Total: **Correct: 90** **Incorrect: 20**

b. Dynamic prediction with one bit (cont'd):

ii) In the beginning instructions are NOT in the BHT, or the initial decision is NOT to take the branch

BNZ LOOP1: There are mispredictions in the first and last iterations.
Other predictions are correct.
Correct: 8 Incorrect: 2

BNZ LOOP2: There are mispredictions in the first and last iterations.
Other predictions are correct.
Correct: 10x8 = 80 Incorrect: 10x2 =20

Total: **Correct: 88** **Incorrect: 22**

c. Dynamic prediction with two bits:

i) Assumption: In the beginning, instructions are in the BHT, and the initial decision is to **take the branch**, prediction bits are 11.

BNZ LOOP1: There is a misprediction only in the last iteration (exit).
Other predictions are correct.
Correct: 9 Incorrect: 1

BNZ LOOP2: There is a misprediction only in the last iteration (exit).
Other predictions are correct.
Correct: 10x9 = 90 Incorrect: 10x1 = 10

Total: **Correct: 99** **Incorrect: 11**

c. Dynamic prediction with two bits (cont'd):

ii) In the beginning, instructions are NOT in the BHT

In the first run of the BNZ instructions, since the target address is unknown, next instructions in sequence (not the target of the branch) are fetched into the pipeline.

Hence, there is a misprediction in the first iteration.

After the CPU has decided to branch and the target address has been calculated, information about the BNZ is stored in the BHT, and prediction bits are set to 11.

BNZ LOOP1: There are mispredictions in the first and last iterations.
Correct: 8 Incorrect: 2

BNZ LOOP2: In the first run, there are mispredictions in the first and last iterations.
After the first run the decision is still "branch will be taken".
Therefore, in the 2nd - 10th runs, there will be a misprediction only in the last iteration.
Correct: 8 + 9x9 = 89 Incorrect: 2 + 9x1 = 11

Total: **Correct: 97** **Incorrect: 13**