

Unit 1.1: Concept of Computer Architecture

1.- Concept of Computer Architecture:

- Defines the hardware of a computer attending to its requirements.
- **Instruction Set Architecture:** everything that must be known by assembler programmers.
- **Processor organization:** logical elements enabling the execution of instructions.
- **Implementation.**
- The three levels are dependent, an engineer must compare performance, cost and other features and select.

2.- Computer Requirements:

- Type of required computer, degree of compatibility, OS requirements and market standards.

3.- Technology, power consumption and cost:

- **Transistor size:** the number of transistors grows quadratically and the speed linearly with the reduction of their featured size.
- **Propagation delay:** as feature size is reduced, R and C increases, reducing the fraction of the chip reachable in a single cycle.
- **Power consumption and heat dissipation:** reduce supply voltage, but a minimum is required, as the feature size decreases the leakage current and the static power increase.
- **Implications:** Microprocessor power distribution, heat dissipation and cooling and new materials to reduce the leakage current.
- **Cost:**
 - **Factors decreasing the cost:** Learning curve (reduction of cost over time) and Sales volume (Doubling sales decreases costs).
 - **Design costs:** Factory costs and Size of design team is inversely proportional to feature size.

4.- Evaluation of performance, Architectural improvements:

- **RISC architecture:** simple instructions running fast.
- **Pipelining:** decomposition of the instruction cycle in stages.
- **ILP exploitation:** out-of-order execution of instructions.
- **Parallelism:**
 - **DLP:** Data-Level Parallelism, single operations on multiple data.
 - **TLP:** Thread-Level Parallelism, different tasks running in parallel.

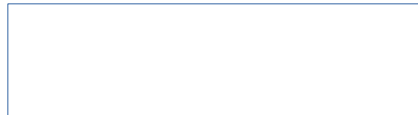
Unit 1.2: Performance Evaluation

1.- Performance definition:

- **Time and Throughput:**
 - **User point of view:** Response time or **Execution time**, time to complete a task.
 - **System manager point of view: Throughput**, operations/executions completed per time unit.

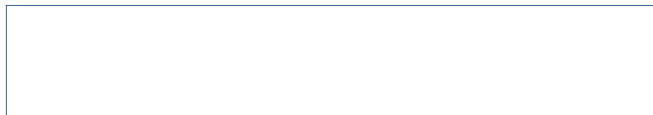


- **Comparisons:**
 - When two computers are compared we take the slowest one as a reference, when we compare designs a computer is selected as reference.
 - A workload must be selected in order to measure the respective performance under similar conditions.
 - **Speedup:** X is S times faster than Y or X is n% faster than Y

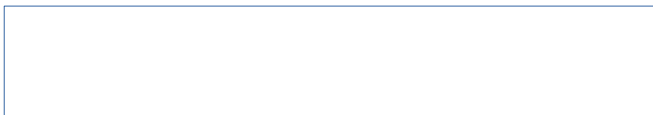


2.- Quantitative Principles of Computer Design:

- **Execution time equation:**
 - All three parameters are related, it is not possible to reduce one without affecting the others.



- **Amdahl's law:**
 - Defines how changes in part of the system affect the whole, F is the fraction of time affected by the improvement and S is the applied speedup.



3.- Measuring Performance:

- **Workloads:**
 - Performance is measured with a program or collection relevant to the user.
 - **Options:**
 - **Real programs:** best option.
 - **Kernels:** fragments of code extracted from real programs.
 - **Toy benchmarks:** simple program with well-known execution results.
 - **Synthetic benchmarks:** programs written to represent the average program typically run in a system.

- **Benchmark suites:**
 - **Components:** typically kernels and non-interactive programs
 - **Updating:** programs in a suite must represent always the typical task run by users.
 - **Reproducibility:** Measures must be reproducible and hardware/software details clearly defined.
- **Comparison of computers:**
 - **Total execution time:**
 - **Arithmetic mean:**
 - **Sum of weighted execution times:** W_i is the frequency of program i in the workload
 - **SPEC:** geometric mean of execution times normalizer wrt a reference machine, R times faster than the reference:

4.- Other performance metrics:

- **MIPS: Millions of Instructions per Second**

- Intuitive and proportional to performance.
- Does not account number of executed instructions
- Depends on considered programs but they are rarely mentioned.
- Depends on the instruction set.
- May be inversely proportional to performance.

- **MFLOPS: Millions of Floating Point Operations per Second**



- Accounts for operations instead of instructions.
- Cannot be applied to programs that don't carry out floating point operations.
- Depends on the FP instruction set, solved by relying on source code FP operations.
- Different programs execute different FP operations with different cost.

Unit 1.3: Instruction set architecture design

1.- General aspects related to instruction sets:

- **Instruction set design factors:**
 - Instruction sets are interfaces between programs and processors.
 - Instructions rarely used by compilers are useless.
 - Most programs are very simple.
 - Fast execution of simple instructions using simple datapaths.
- **Main features of instruction sets:**
 - **Basic principle:** common case, efficient, uncommon case, correct.
 - **Orthogonality:** simplify generation of code making things independent.
 - **Provision of primitives instead of solutions:** avoid the inclusion of solutions directly supporting high-level instructions, they provide more or less functionality than necessary.
 - **Principle one or all:** either there is only one way to make something, or all ways are possible.
 - Integrate instructions operating on constants for values already known in compilation-time.

2.- Types of instruction sets:

- **Basic Paradigms:**
 - Storage of operands in the CPU, instructions compute on input data and produce a result, which can be stored in memory and CPU, current instruction sets use a general purpose register file.
- **L/S and R-M models:**
 - **RISC L/S Model:**
 - ALU instructions contain three operands and are computed in registers.
 - Load and Store instructions exchange data between registers and memory.
 - Elevated number of simple instructions, easy decoding and similar work.
 - **IA R-M Model:**
 - Instructions have two operands that can be on registers or memory.
 - MOV instruction exchanges data.
 - Less instructions but more complex decoding and diverse amount of work.

3.- Registers and Operand types:

- **RISC** have big number of registers of the same length, instructions use the whole content.
- **IA** have few registers that are adapted to the available data types.

4.- Instruction encoding:

- **Encoding strategies:**
 - **Fixed format:** all instructions are encoded using the same number of bits, easier fetching and decoding but sometimes waste of bits.
 - **Variable format:** instructions use only the required amount of bits, space is optimized but decoding and fetching is more complex.
 - The number of bits of the format limits the number of available variants.
 - **Number of instruction formats:**
 - **One format:** correspondence between format bits and fields is fixed, easier decoding.
 - **Multiple format:** each format can have different fields and establish a correspondence, better bit adjustment.

5.- Memory addressing:

- **Address interpretation:**
 - Physical units contain $W = 2^w$ addressable units.
 - Words referred through the address of their least significant byte.
 - **Alternatives:** Little endian (first byte is the LSB), Big endian (first byte is the MSB).
- **Alignment:**
 - Instruction sets provide access to units of 2^w bytes.
 - RISC Aligned access: the address of the object of 2^i bytes is multiple of 2^i .
 - IA Non-aligned access: there is no restriction sometimes demands more than one physical access to memory.
 - Sophisticated addressing modes reduce the number of instructions but increase the complexity of hardware.

6.- Control flow:

- **Control instructions:**
 - 5/6 of branches are typically taken in a regular program.
- **Addressing modes:**
 - **PC-relative:** destination usually near the current instruction, bits to encode the relative displacement.
 - **Indirect link:** when the branch is unknown at compilation time, statements selected one over several alternatives.
- **Branch conditions:**
 - **Conditions codes:** a state defined by the IS is modified by the last ALU operation or flags can be checked to see condition codes, requires space and can be problematic.
 - **Explicit check:** result of operation is checked using specific instructions.

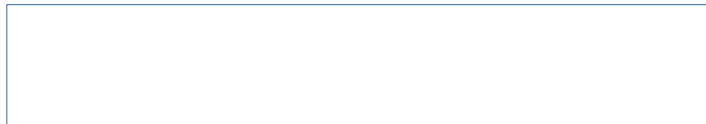
7.- SIMD Instructions:

- **Packed Data:**
 - SIMD instructions work on registers holding packed data.
 - The number of elements included in one register is n , its length divided the data-type length.
 - SIMD instructions perform n operations simultaneously.
- **SAXPY with SIMD instructions:**
 - Each vector component takes 32 bits, registers have 128 bits so each contain 4 components, the SIMD `a_reg` register is initialized with 4 copies of A , we read 4 components of each vectors and store them in `x_reg` and `y_reg`, a single instruction performs four $A * X[i]$ multiplications and another one the four additions, the another instructions updates the contents of Y in memory, so we only need $n/4$ iterations.
- **Compiler support:**
 - **Automatic vectorization:** an advanced compiler extracts that parallelism from the source scalar code.
 - **Use of SIMD datatypes:** programmer defines variables of interest and uses them in the program as common expressions.
 - **Intrinsic functions:** there is a SIMD function library to enable the insertion of the required vector code.

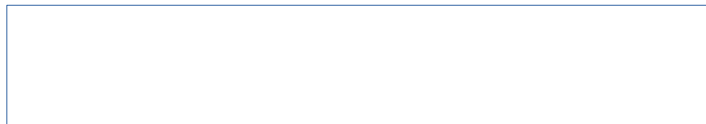
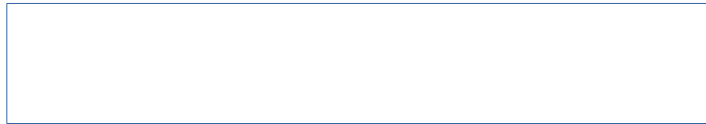
Unit 2.1: Pipelined instruction unit

1.- Concept of pipelining:

- **Pipelining:**
 - A process is decomposed in several subprocesses, each one executed in an autonomous mode concurrently with the rest.
- **Pipelines:**
 - A system is pipelined in k stages, modules processing a subtask and a latch register.
 - **Latches:** keep data stable the time required to cope with its functions.
 - A clock synchronizes the advance of data.
 - **Clock period:**



- **Pipelining improvements:**
 - **Speed-up :**
- **Throughput:** results / unit of time.



2.- Pipelining the instruction cycle:

- **Monocycle vs Multicycle control:**
 - **Monocycle:** the period is the time required to execute the longer instruction, all instructions spend one cycle and there is no need of temporal registers.
 - **Multicycle:** the period is the time required to execute the longer stage, each instruction lasts a different number of periods, need of temporal registers.

3.- Instruction cycle pipelining:

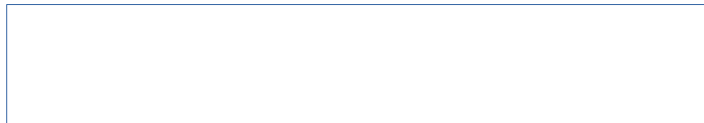
- **MIPS pipelining:**
 - Stages executed by instructions may vary, but we assume that all traverse the same set of stages and some instructions perform nothing in those stages.
- **Hardware requirements:**
 - In a single cycle there are 5 instructions being processed.
 - Cache access time doesn't vary but the required bandwidth is five times bigger.

4.- Control of the pipelined instruction cycle:

- **Control signals:**
 - Required control signals are generated in the ID stage.
 - Identifier of destination register follows same path as data.
 - PC management logic done in IF.

5.- Hazards:

- **Concept and classification:**
 - **Hazard:** situation leading to the execution of instructions generating incorrect results.
 - **Hazard types:** data, control and structural.
- **Hazard solutions:**
 - **Stall insertion:** stop instructions originating a hazard and following ones, during this cycles no instruction is fetched.



- **Datapath modification:** modify the hardware to dynamically solve hazards, reduce the number of stalls but a complete solution is sometimes impossible.
- **Compiler modification:** avoid the generation of certain sequence of instructions, loss of binary compatibility when datapaths accept any sequence of instructions.

6.- Data hazards:

- **Stall insertion:**
 - Delay those operations originating a hazard, insert a nop internal signal in the latches.
 - 3 stalls for each two dependent and consecutive instructions, significant loss of performance.
- **Datapath modifications:**
 - Reduction of the number of instructions involved in a hazard.
 - **Multiport register file** supporting simultaneously read and write.
 - **Modification of the access time to the register file**, write on first semicycle and read on the second.
 - The penalty for dependent consecutive instructions is of 2 stalls, WB and ID can happen at the same time.
 - **Forwarding technique:** add a bus in the ALU output (MEM) to its inputs (EX) and control logic (multiplexers in the ALU input), create a short-circuit from MEM to EX.
- **Load-dependent and consecutive instructions:**
 - If the result is obtained in a stage that will be evaluated temporally after the one we need it, in addition of a short-circuit we need to insert stalls.
- **Compiler modifications:**
 - **Delayed load technique:** compiler ensures that after a load there are no instructions reading the loaded register, it rearranges the code inserting useful and independent instructions after the load, or inserting nops.
 - **Load delay slot:** number of instructions following a load that should not read from the loaded register, in the worst case the performance is the same as when using stalls but without additional circuitry but the code size increases and we lose binary compatibility.

7.- Control hazards:

- **Causes:**
 - Branch instructions do not modify the PC in the MEM stage.
- **Stall insertion:**
 - Insert stalls whenever a branch is decoded, disabling IF during 3 cycles sending NOP to ID.
- **Datapath modification:**
 - **Predict-not taken:** if the branch is finally taken, the three instructions following the branch will be canceled, they must not modify the computer state.
 - **Predict taken:** once the destination address is known new instructions are fetched, only useful when the destination address is known before the branch condition.
 - **Reducing branch latency:** reduce the number of cycles between fetching the branch instruction and computing the destination.
 - **Impact on the clock period:** the ID stage may take too long in the branch condition is not simple.
- **Compiler modifications:**
 - **Delayed branch:** compiler places after branches valid instructions or nops, they will always be executed.
 - **Branch delay slot:** number of instructions that must be placed after the branch, equal to the branch latency.
 - In the worst case the performance is as bad as with stalls but without additional circuitry but we increase the code and loss binary compatibility.

8.- Structural hazards:

- **Causes:**
 - Hardware does not allow all possible combinations between instructions in the unit.
- **Solutions:**
 - **Datapath modifications:** replicate resources in order to enable the desired combination of instructions (Harvard architecture).
 - **Stall insertions:** delay instructions generating the hazard.

9.- Exceptions:

- **Concept and classification:**
 - Asynchronous or Synchronous if the event is raised at the same location in every execution.
 - User-driven or Raised to the user.
 - Maskable or Unmaskable.
 - During one instruction or Between instructions.
 - Continue program or End program.
- **Exceptions in conventional computers:**
 - i1, i2, i3 – handler – i3, i4...
- **Exceptions in pipelined computers:**
 - There are instructions following the one raising the exception.
 - The sequence of executed instructions is incorrect since the PC is only kept until ID.
- **Precise exceptions in pipelined instruction units:**
 - Instructions preceding the one generating the exception finish correctly.
 - The instruction raising the exception and the following ones are aborted.
 - After the handler it is possible to restart the program from the instruction originating the exception.
- **Implementation of precise exception in MIPS:**
 - Instructions must reach the last stage of the pipeline in order.
 - Each instruction entering the unit is related to a register with as many bits as stages that travels through the pipeline.
 - If an exception is raised, the bit of the stage becomes 1 and the instruction becomes a nop.
 - In the last stage, if any register bit is set, following instructions become nop and a trap is generated in the IF.
 - The PC of the instruction raising the exception is stored, and if we use delay branch, also store the PC of delay slot plus one.
 - Exception handler takes control, and once it is executed PCs are restored following execution from that point.

Unit 2.2: Multicycle Units and Static Instruction Scheduling

1.- Multicycle operations:

- **Problem:**
 - Some integer instructions and floating point ones require extra time in EX.
- **Solution:**
 - Enable Ex stage of complex operations to take various clock cycles.
- **Multiple operators:**
 - New specialized operators are added to cope with new instructions, during ID the instruction is issued to the adequate operator.
- **Types of multicycle operators:**
 - Can be conventional or pipelined.
 - **Latency or Evaluation time:** time required to obtain the first result.
 - **Initiation Rate (IR):** reverse of the time between results, 1 if pipelined.
- **Pipelined MIPS including new operators:**
 - MEM is empty in multicycle operations.
 - New inter-stage registers required and a multiplexer manages WB.
- **Structural hazards for using units with $IR < 1$:**
 - **Problem:** simultaneous instructions in the same operator.
 - **Solution:** the second instruction must be stalled in ID and the rest in IF.
- **Structural hazards derived from simultaneous register file writing:**
 - **Problem:** several instructions in WB.
 - **Solutions:**
 - Separate integer from floating point register files to reduce the number of simultaneous accesses.
 - Increase the number of ports.
 - Stall insertion: ID checks whether an instruction writes to the register file at the same time as other does.
- **Integer and floating point independent register files:**
 - **Advantages:** structural hazards are reduced, register number and register file bandwidth are duplicated.
 - **Drawbacks:** sometimes is necessary to exchange information between register files and the number of registers of each type is limited.
- **Structural hazards:**
 - **RAW (Read After Write):** when an instruction produces a result required by a following one, insert stalls until a short-circuit can be applied.
 - **WAW (Write After Write):** when two close instructions write to the same register, stall to ensure that registers are written in order.
- **Exception management:**
 - The presence of multicycle operations can alter the instruction execution order, when an exception occurs, some of the instructions following the one triggering the exception may have already finished.

2.- Types of dependencies:

- **Instruction level parallelism (ILP):**
 - Potential overlapping in the execution of instruction sequences.
 - Relies on the independence among the considered instructions.
- **Dependencies between program instructions:**
 - Two instructions are independent if they can be simultaneously executed without any problem, they can be reordered.
- **Data dependency:**
 - Given two instructions I and j, there is dependency if I produces a result used by j or by an instruction k that produces a result used by j, shared data can be stored in registers or memory.
- **Name dependency:**
 - When two instructions use the same register or memory location, but there is no data flow between them.
 - **Anti-dependency:** when an instruction requires a value that is later updated.
 - **Output dependency:** when the ordering of instructions will affect the final output value of a variable
- **Control dependencies:**
 - An instruction has a control dependency with a preceding one if the latter's outcome conditions the former's execution.
 - Every instruction in a program has a control dependency with some branch.

3.- Improving ILP:

- **Basic block concept:**
 - Increase the ILP of instructions under execution in the pipelined unit.
 - Sequence of instruction between branches are basic blocks, we need to determine the level of parallelism that can be extracted from the execution of the instructions in each basic block.
- **Is there enough ILP in a basic block?**
 - 15% of instructions are branches and there are 6 or 7 instructions per basic block.
 - Instructions in a basic block usually exhibit dependencies among them.
 - Solution: exploit ILP among multiple basic blocks.
- **Static instruction scheduling:**
 - The compiler reorders/modifies the code to increase ILP reducing existing dependencies and their effects
 - **Loop unrolling:** moves dependent instructions away from each other.
 - **Software pipelining:** reorders the code in order to move dependent instructions away from each other.

4.- Loop Unrolling:

- **Basic idea:**
 - The loop code is replicated several times, reducing the number of iterations executed.
 - Reduces the overhead produced by loop control instructions.
- **Optimization:**
 - It is necessary to perform register renaming to eliminate name dependencies.
 - Dependent instructions are separated by inserting as many instructions between them as the number of required stalls.

5.- Software Pipelining:

- **Basic idea:**
 - Transform a loop with dependent instructions and independent iterations into another loop with dependent iterations and independent instructions.
 - Simulates the behavior of a pipelined unit.

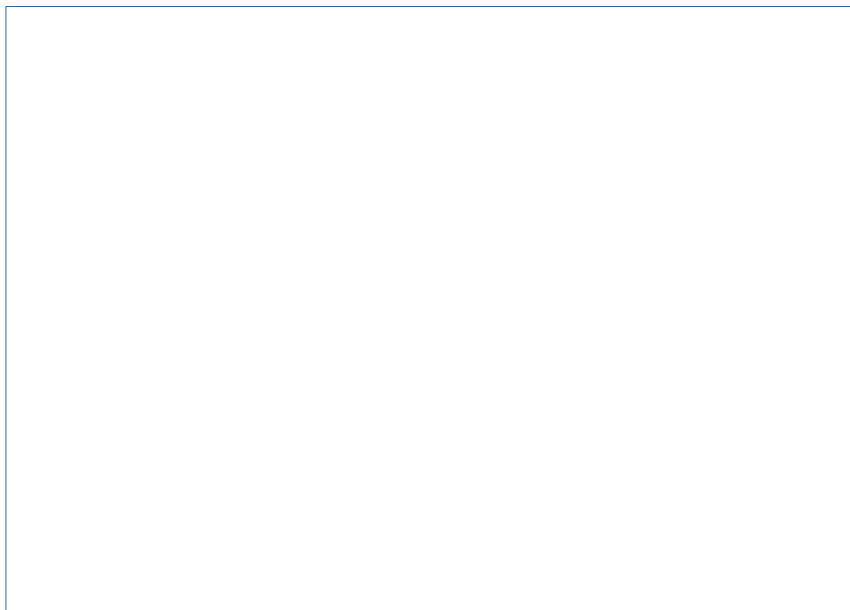
Unit 2.3 Dynamic Branch Prediction

1.- Introduction:

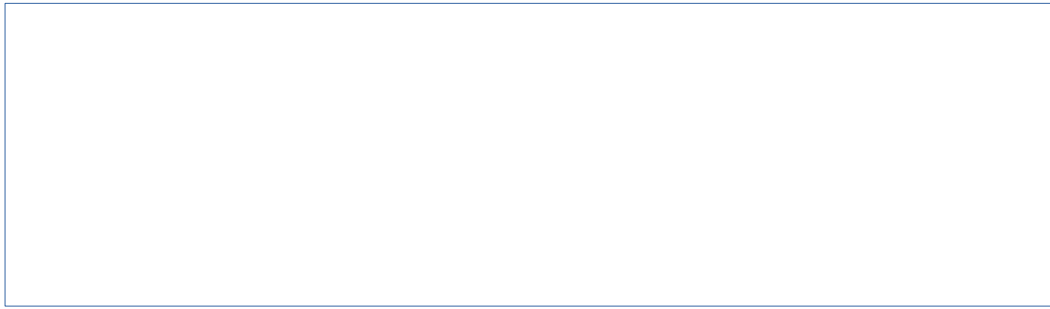
- **Motivation:**
 - There is not enough ILP in a basic block
 - It is necessary to obtain the result of a branch as soon as possible.
- **Goal:**
 - Predict the behavior of branches in order to have instructions fetched by the time the final result of the branch becomes available.

2.- Branch Prediction Buffers (BPB):

- **Mechanism:**
 - A table indexed with the least significant bits of the branch address.
 - Each entry contains the prediction for that branch instruction.
- **Implementation:**
 - **Table location:**
 - Small fast memory accessed in parallel with cache during IF.
 - Adding prediction bits to each instruction cache block.
 - **Predictor:**
 - Finite state automaton.
 - State changes according to the actual behavior of the branch.
 - Prediction depends on the state.
- **Two branches requiring the same entry:**
 - Prediction may fail if two branches have the same least significant bits, so solve this we need to increase the table size and add more bits for the PC index.
- **Branch prediction in loops:**
 - If the loop contains n iterations, the branch is taken n-1 times.
 - A one-bit predictor will fail in the first and last iteration.
- **Two-bit predictors:**
 - The automaton has four two-bit states.
 - A prediction must fail twice before being modified.
- **Two-bit predictor with hysteresis:**



- **N-bit predictor with saturation:**



- Each entry has a value of n bits (0 to $2^{(n-1)}$).
 - Prediction must fail $2^{(n-1)}$ times before being modified.
- **Two-level or correlating predictors:**
 - In addition to the behavior of the current branch they consider the behavior of other branches.
 - A predictor (g: global, l: local) uses the behavior of the last g branches to choose among 2^g l-bit predictors.
- **Hybrid predictors:**
 - Each predictor is suitable for different patterns, by combining them and applying the most suitable one it is possible to increase the accuracy.
 - **Selection mechanism:** use the predictor that provided the best result up to now.
 - **Implementation:** table of saturating counters indexed by the branch address:
 - Predictor 1 | Predictor 2 | Counter
 - The most significant counter bit selects the predictor to use.

3.- Branch Target Buffers (BTB):

- **How it works:**
 - **Completely associative table with three fields:**
 - **Index:** Branch instruction address.
 - **Prediction:** Taken or Not taken.
 - **Address:** Branch target address.
- **Events at stages of the instruction unit:**
 - **IF:** if the instruction is on the BTB table and the prediction is taken, start fetching instructions during next cycle from the address provided by the table.
 - **ID:** if the instruction is a branch, compute the address and condition, if the instruction is in the table and the prediction is different than the condition, abort incorrectly fetched instructions, update the prediction and start fetching from correct address, if the instruction was not in the table add an entry.
- **Comparison with predict not taken:**
 - The effective behavior depends on the accuracy, we need a high predictor hit rate.
 - As branch conditions may depend on previous long-latency instructions and may take long to complete, BTB would be much better.

- **Implementation:**

- The table stores the complete address of the branch, a hit only occurs if the instruction is a branch for sure.
- It is not mandatory to implement a fully associative table, it can be set-associative.
- The BTB and the predictor can be split:
 - **BHT:** stores prediction bits.
 - **BTB:** stores the target address of the last taken branches.
 - If the prediction of the BHT is taken and there is a hit in the BTB, it starts fetching from the target address.