Superscalar:

目的：使得一个 cycle 可執行多个指令，exploit ILP

降低 program execution time 方式：

a. 減少各別指令的 latency

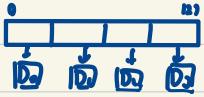
b. 增加 throughput

Parallel instruction processing requires: the determination of the dependence relationships between instructions, adequate hardware resources to execute multiple operations in parallel, strategies to determine when an operation is ready for execution, and techniques to pass values from one operation to another. When the effects of instructions

2.

階段：

I. 一次 fetch 和 decode 一个 instruction stream (multiple instructions)



II. conditional branch prediction 來確保 instruction stream 不會因 branch jump 中斷

III. 分析 指令間的 dependency

IV. 分派指令給 functional unit，平行執行

根據 availability of operand data.

V. commit instru. 使其執行順序和原 instruction stream 相同

為了 parallel instr. processing 需 implement

1) Instruction fetch strategies that simultaneously fetch multiple instructions, often by predicting the outcomes of, and fetching beyond, conditional branch instructions.
2) Methods for determining true dependences involving register values, and mechanisms for communicating these values to where they are needed during execution.
3) Methods for initiating, or *issuing*, multiple instructions in parallel.
4) Resources for parallel execution of many instructions, including multiple pipelined functional units and memory hierarchies capable of simultaneously servicing multiple memory references.
5) Methods for communicating data values through memory via load and store instructions, and memory interfaces that allow for the dynamic and often unpredictable performance behavior of memory hierarchies. These interfaces must be well matched with the instruction execution strategies.
6) Methods for committing the process state in correct order; these mechanisms maintain an outward appearance of sequential execution.

解決 control hazard:

As a static program executes with a specific set of input data, the sequence of executed instructions forms a *dynamic* instruction stream. As long as instructions to be executed are consecutive, static instructions can be entered into the dynamic sequence simply by *incrementing* the program counter, which points to the next instruction to be executed. When there is a conditional branch or jump, however, the program counter may be *updated* to a nonconsecutive address. An instruction is said to be *control dependent* on its preceding dynamic instruction(s), because the flow of program control must pass through preceding instructions first. The two methods of modifying the program counter—incrementing and updating—result in two types of control dependences (though typically when people talk about control dependences, they tend to ignore the former).

The first step in increasing instruction level parallelism is to overcome control dependences. Control dependences due to an incrementing program counter are the simplest, and we deal with them first. One can view the static program as a collection of *basic blocks*, where a basic block is a contiguous block of instructions, with a single entry point and a single exit point [1]. In the assembly code of Fig. 1, there are three basic blocks. The first basic block consists of the five instructions between the label L2 and the *ble* instruction, inclusive, the second basic block consists of the three instructions between the *ble* instruction, exclusive, and the label L3, and the third basic block consists of the three instructions between the label L3 and the *blt* instruction, inclusive. Once a basic block has been entered by the instruction fetcher, it is known that all the instructions in the basic block will be executed eventually. Therefore, any sequence of instructions in a basic block can be initiated into a conceptual *window of execution*, en masse. We consider the window of execution to be the full set of instructions that may be simultaneously considered for parallel execution. Once instructions have been initiated into this window of execution, they are free to execute in parallel, subject only to data dependence constraints (which we will discuss shortly).

```
for (i=0; i<last; i++) {
  if (a[i] > a[i+1]) {
    temp = a[i];
    a[i] = a[i+1];
    a[i+1] = temp;
    change++;
  }
}
```

(a)

```
L2:
    move    r3,r7       #r3->a[i]
    lw      r8,(r3)     #load a[i]
    add     r3,r3,4     #r3->a[i+1]
    lw      r9,(r3)     #load a[i+1]
    ble     r8,r9,L3    #branch a[i]>a[i+1]

    move    r3,r7       #r3->a[i]
    sw      r9,(r3)     #store a[i]
    add     r3,r3,4     #r3->a[i+1]
    sw      r8,(r3)     #store a[i+1]
    add     r5,r5,1     #change++
L3:
    add     r6,r6,1     #i++
    add     r7,r7,4     #r4->a[i]
    blt     r6,r4,L2    #branch i<last
```

(b)

← Basic block

basic block 保證了 block 內之為 sequential executed 且當 block 中第一个指令開始執行，則最後一个指令必會被執行到

∴ 以 basic block 為 window of execution，basic block 內可平行執行，除非有 data dependency

但要盡可能地 exploit parallelism，需利用 Branch prediction / Speculation 來猜測 branch taken or not，若猜錯再來修復

program 中指令執行如下圖：

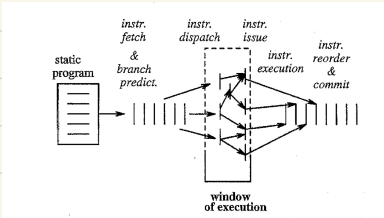


Fig. 3. A conceptual figure of superscalar execution. Processing phases are listed across the top of the figure.

microarch of superscalar

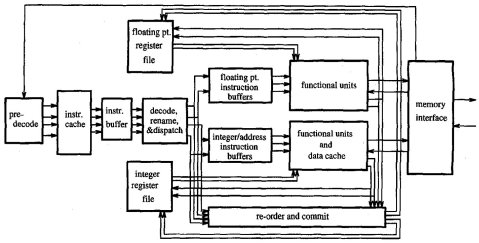


Fig. 4. Organization of a superscalar processor. Multiple paths connecting units are used to illustrate a typical level of parallelism. For example, four instructions can be fetched in parallel, two integer/address instruction scan issue in parallel, two floating point instructions can complete in parallel, etc.

phase 1. instr. fetching & branch prediction

PC 中有多个指令位址，∴ instr. cache 要設計成 block size 為數个 instruction size

否則很容易 cach miss，造成 performance 的損失

又∵需要高的 bandwidth (同時 fetch 多个指令) ∴會 split cache

同時，high-bandwidth instruction fetch 會利用 prefetch 來減少 miss rate

再來 branch 指令的 redirection 會造成 delay

Processing of conditional branch instructions can be broken down into the following parts:

1) recognizing that an instruction is a conditional branch,
2) determining the branch outcome (taken or not taken),
3) computing the branch target, and
4) transferring control by redirecting instruction fetch (in the case of a taken branch).

(1). 可以透過 predecode logic，作為輔助判別是否為 branch instruction 之 information 存在 instr. cache 中 predecode branch instruction opcode：op0~op6 ⟶ ! 甚至 identify 其它 instru.

(2). 當 branch 指令 fetch 時，可能 data, operand 尚未準備好 (ex. 其它 instr. 有 dependency)
∴ 利用 predictor 來 predict

① static branch prediction：利用 complier profiling 的 information，會為 static binary

② dynamic branch prediction：在 run-time 利用先前執行紀錄來 predict
需 HW 支援，branch prediction table

(3). 計算跳躍目的位址 - 般需 adder
而 ∵ ISA 定義之 branch addressing mode - 般為 PC + offset
∴ 不需 read register
可再利用 branch target buffer 加速此 step.

phase II. Instruction Decoding, Renaming, and Dispatch

從 instruction buffer 移除指令，並 detect 指令間的 data & control dependency
包含：① true data hazard detection (RAW)
② anti dependency & output (WAR, WAW)
dispatch 指令至各 functional unit 的 buffer