07.Instruction Level Parallelism

Definition of ILP

Parallelism between instruction, collision between uncorrelated instruction
ILP = Potential overlap of execution among unrelated instructions
Overlapping is possible if:

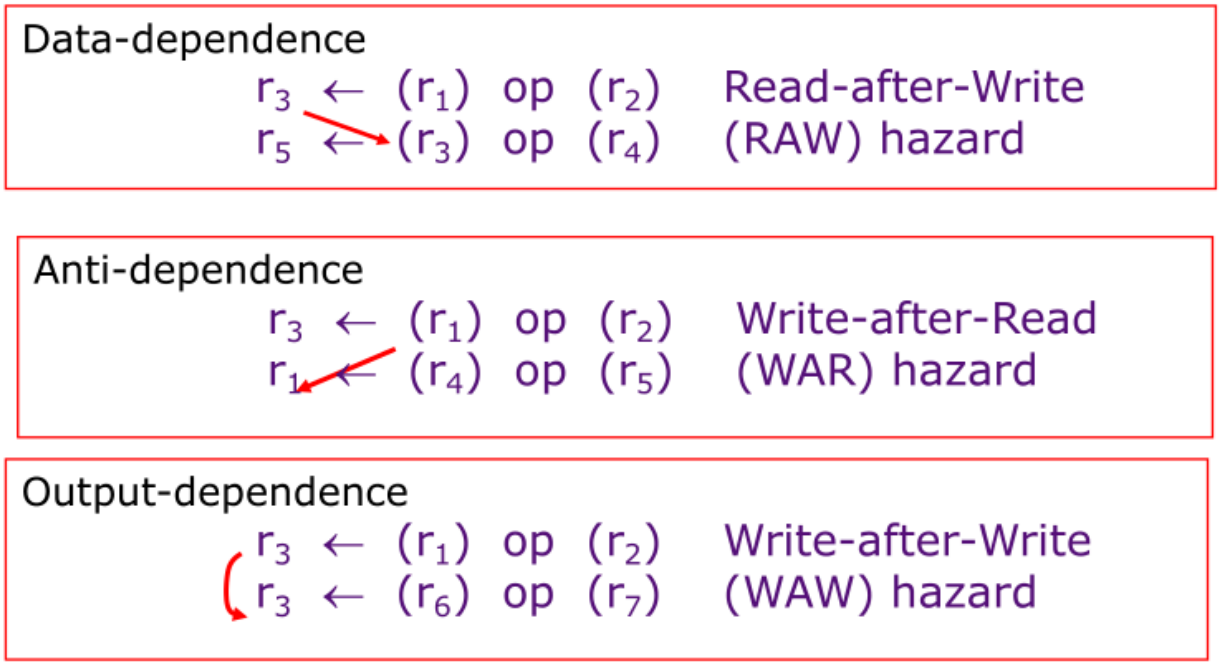
- No Structural Hazards: manage with the architecture of the processor
- No RAW, WAR of WAW Stalls
- No Control Stalls

Pipeline performance

Ideal pipeline CPI: measure of the maximum performance attainable by the implementation
Pipeline CPI = Ideal pipeline CPI + Structural Stalls

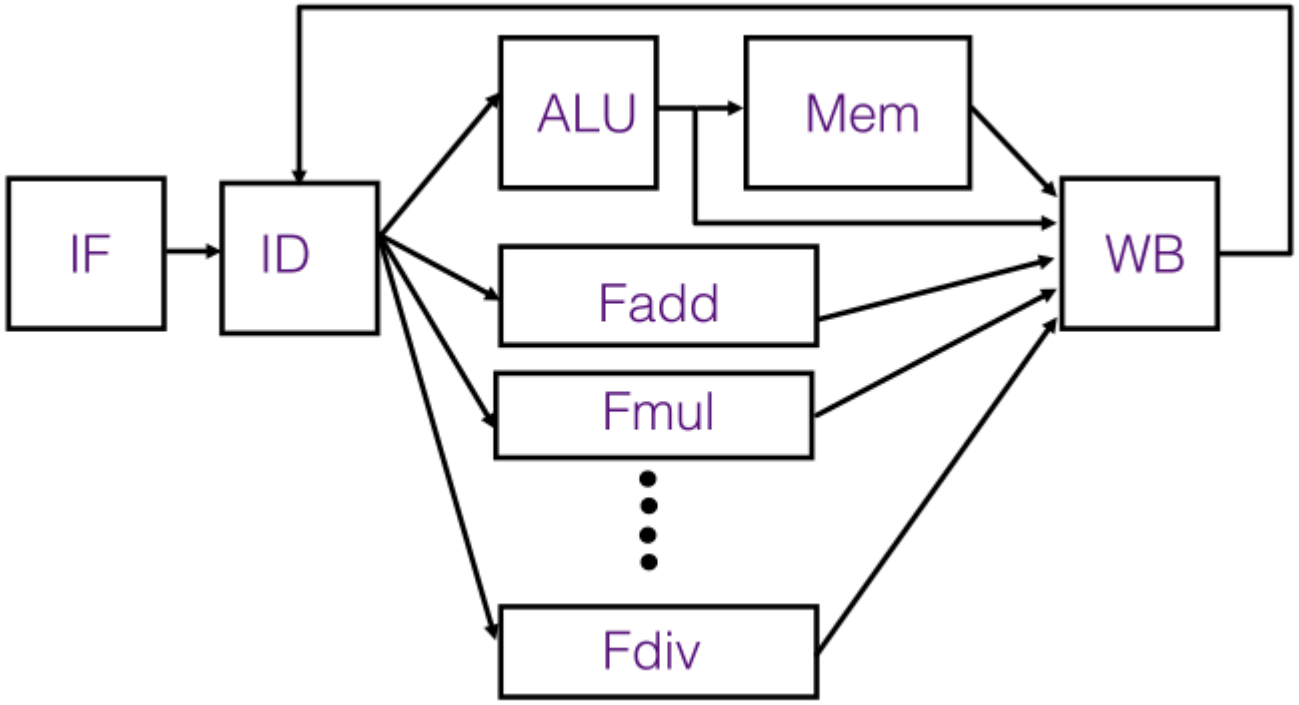
- Ideal pipeline CPI: measure of the maximum performance attainable by the implementation
- Structural hazards: HW cannot support this combination of instructions
- Data hazards: Instruction depends on result of prior instruction still in the pipeline
- Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches, jumps, exceptions)
  If you can lower one of this parameter you can lower the Pipeline CPI
  The longer the Pipeline the greater the impact of the hazard. Pipelining helps instruction bandwidth not latency.

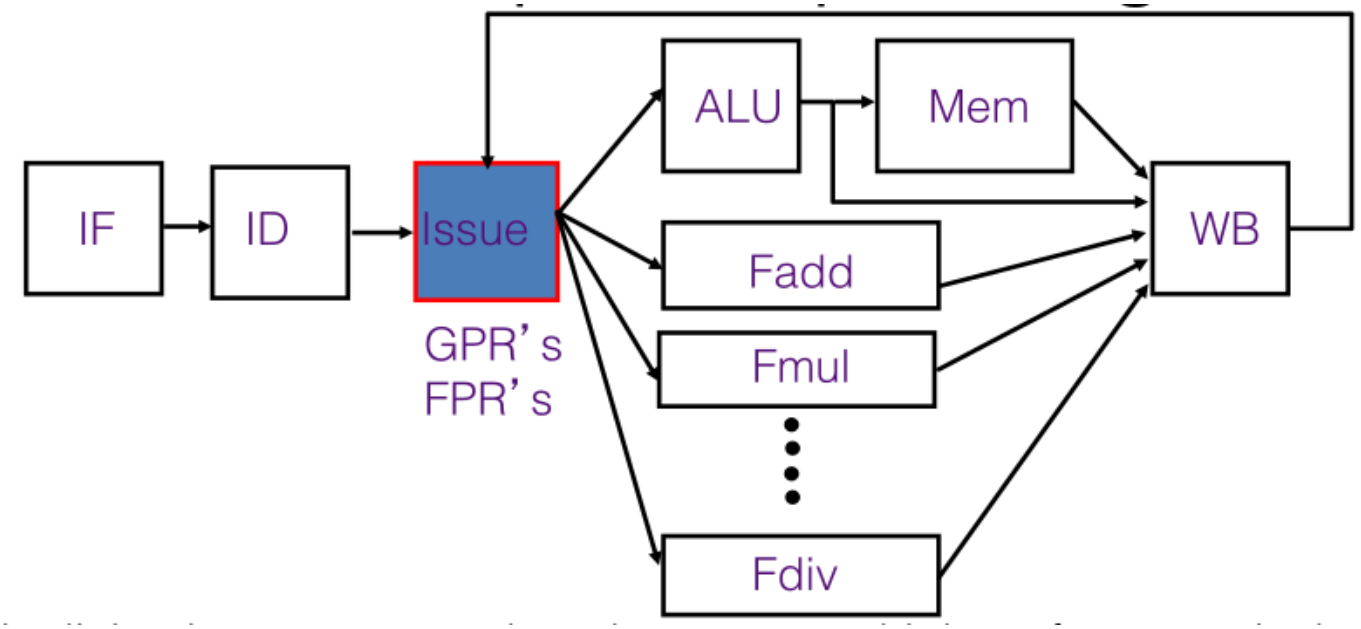
Review: Types of Data Hazards



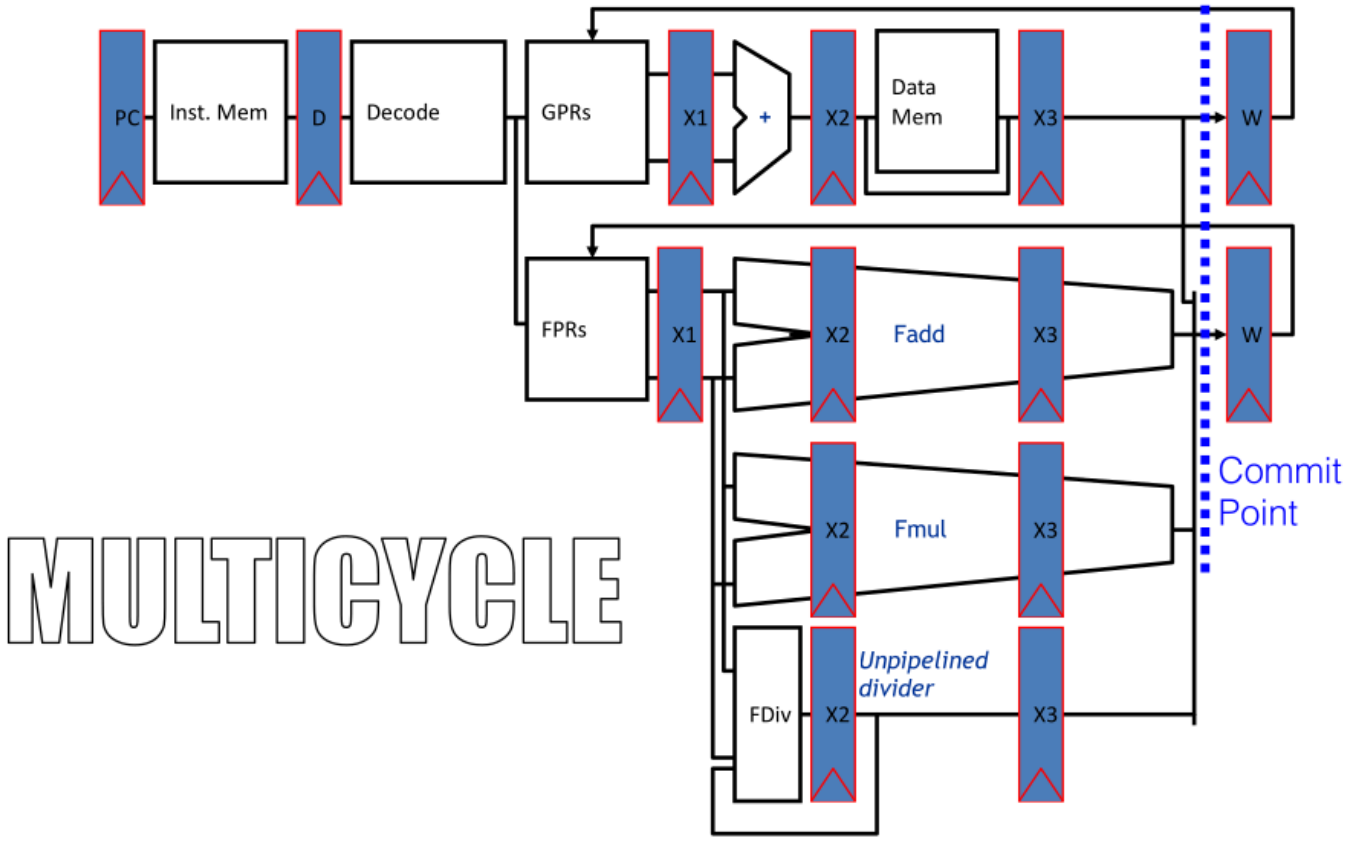
In general the result is that you cannot change the order in which you write in a registry.

Complex Pipelining





In this case we are adding floating and integer numbers. We want to handle different registers. In general the Mem is connected to the ALU. We want high performance and low latency. But we have the long latency or partially pipelined floating-point units, multiple function and memory units, memory systems with variable access time or precise exception. We have to manage this and on stage level we have to pipeline also intraoperation, we use a multicycle architecture.



When you decode the operation you decode the type of operation and send the operation to the general or the floating points pipeline.
Structural conflicts at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle. Structural conflicts at the write-back stage due to variable latencies of different functional units. Out-of-order write hazards due to variable latencies of different FUs.

We can manage this part with delay writeback so all operations have same latency to WB stage:

- Write ports never oversubscribed (one inst. in & one inst. out every cycle)
- Instructions commit in order, simplifies precise exception implementation
  This obviously introduce delays. You can try to resolve the problem without bypassing but it need a complex analysis.

When is it Safe to Issue an Instruction?

The following checks need to be made before the Issue stage can dispatch an instruction:

- Is the required function unit available?
- Is the input data available? ---> RAW?
- Is it safe to write the destination? ---> WAR? WAW?
- Is there a structural conflict at the WB stage?

Assumptions

- All functional units are pipelined
- Registers are read in Issue stage: we consider that a register is written in the first half of the clock cycle and read in the second half
- No forwarding
- ALU operations take 1 clock cycle
- Memory operations take 2 clock cycles (includes time in ALU)
- FP ALU operations take 2 clock cycles
- The Write Back unit has a single write port
- Instructions are fetched, decoded and issued in order
- An instruction will only enter the ISSUE stage if it does not cause a WAR or WAW hazard. The instruction is held if it is not safe to execute.
- Only one instruction can be issued at a time, and in the case multiple instructions are ready, the oldest one will go first (aka ISSUE seen as an infinite buffer, RAW can enter and will be solved here)
  If we can reduce the number of CC per instruction we increase the bandwidth.
  To reach higher performance (for a given technology) – more parallelism must be extracted from the program.
  Dependences must be detected and solved, and instructions must be ordered (scheduled) so as to achieve highest parallelism of execution compatible
  with available resources. This is a critical operation as dependencies are critical.
  Determining dependences among instructions is critical to defining the amount of parallelism existing in a program, but is also difficult as you could need an advanced analysis to the code.
  If two instructions are dependent, they cannot execute in parallel: they must be executed

in order or only partially overlapped.
Three different types of dependences:

- Name Dependencies
- Data Dependencies (or True Data Dependencies)
- Control Dependencies

Name Dependencies

Name dependence occurs when 2 instructions use the same register or memory location (called name), but there is no flow of data between the instructions associated with that name.
Two type of name dependencies between an instruction i that precedes instruction j in program order:

- Antidependence: when j writes a register or memory location that instruction i reads. The original instructions ordering must be preserved to ensure that i reads the correct value.
- Output Dependence: when i and j write the same register or memory location. The original instructions ordering must be preserved to ensure that the value finally written corresponds to j.
  Name dependencies are not true data dependencies, since there is no value (no data flow) being transmitted between instructions.
  If the name (register number or memory location) used in the instructions could be changed, the instructions do not conflict.
  Dependencies through memory locations are more difficult to detect (“memory disambiguation” problem), since two addresses may refer to the same location but can look different.
  Register renaming can be more easily done.
  Renaming can be done either statically by the compiler or dynamically by the hardware.

Data Dependences and Hazards

A data/name dependence can potentially generate a data hazard (RAW, WAW, or WAR), but the actual hazard and the number of stalls to eliminate the hazards are a property of the pipeline.

- RAW hazards correspond to true data dependences.
- WAW hazards correspond to output dependences
- WAR hazards correspond to antidependences.
  Dependencies are a property of the program, while hazards are a property of the pipeline.

Control Dependences

Control dependence determines the ordering of instructions and it is preserved by two properties:

- Instructions execution in program order to ensure that an instruction that occurs before a branch is executed before the branch.
- Detection of control hazards to ensure that an instruction (that is control dependent on a branch) is not executed until the branch direction is known.
  Although preserving control dependence is a simple way to preserve program order, control dependence is not the critical property that must be preserved.

Program Properties

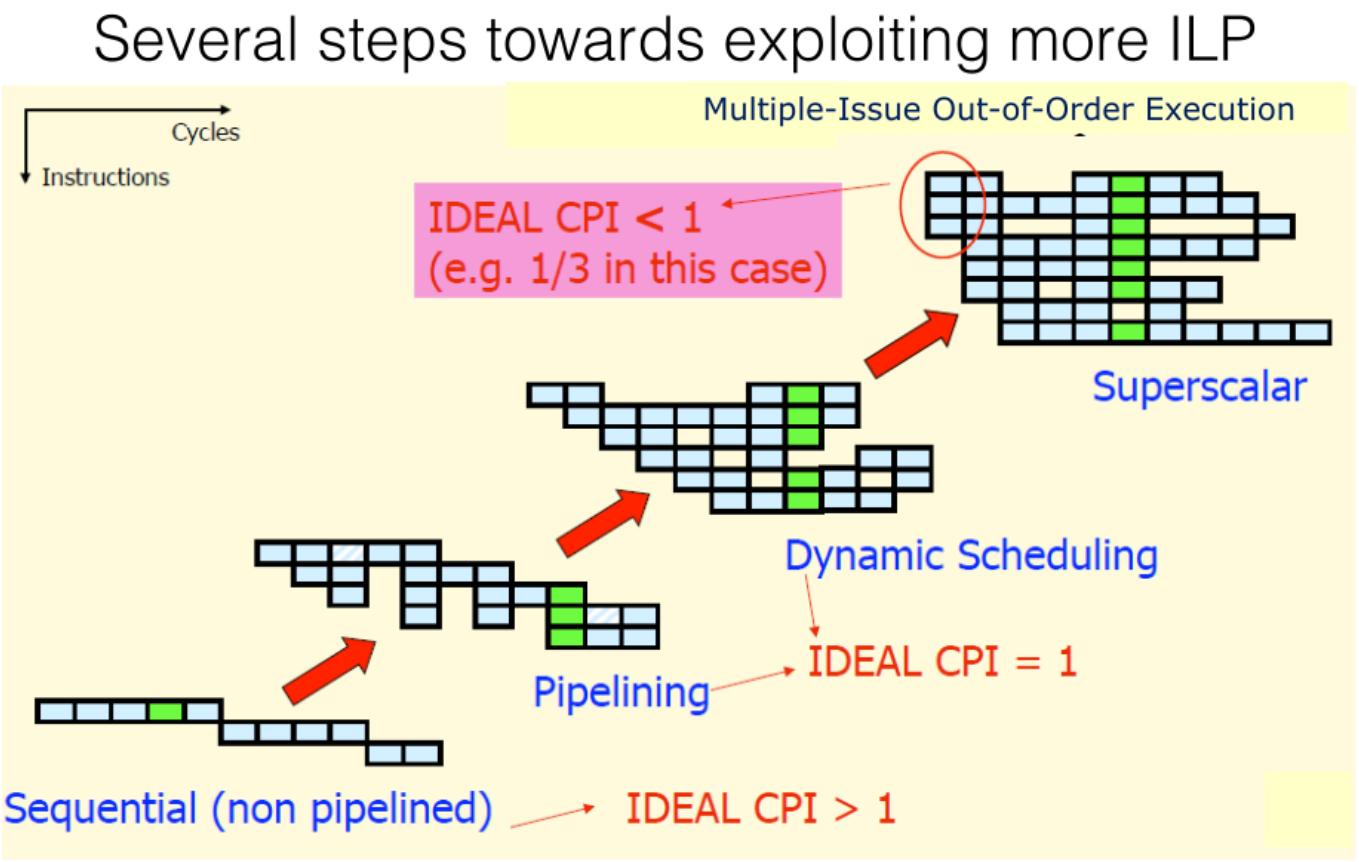
Two properties are critical to program correctness (and normally preserved by maintaining both data and control dependences):

- Exception behavior: Preserving exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program.
- Data flow: Actual flow of data values among instructions that produces the correct results and consumes them.
  Two strategies to support ILP:
- Dynamic Scheduling: Depend on the hardware to locate parallelism
- Static Scheduling: Rely on software for identifying potential parallelism
  Hardware intensive approaches dominate desktop and server markets

Dynamic Scheduling

The hardware reorders the instruction execution to reduce pipeline stalls while maintaining data flow and exception behaviour.
Main advantages:

- It enables handling some cases where dependences are unknown at compile time
- It simplifies the compiler complexity
- It allows compiled code to run efficiently on a different pipeline.
  Those advantages are gained at a cost:
- A significant increase in hardware complexity,
- Increased power consumption
- Could generate imprecise exception
  Basically: Instructions are fetched and issued in program order (in- order-issue)
  Execution begins as soon as operands are available possibly, out of order execution note: possible even with pipelined scalar architectures.
  Out-of order execution introduces possibility of WAR, WAW data hazards.
  Out-of order execution implies out of order completion unless there is a re-order buffer to get in-order completion



Static Scheduling

Compilers can use sophisticated algorithms for code scheduling to exploit ILP (Instruction Level Parallelism).
The amount of parallelism available within a basic block, a straight-line code sequence with no branches in except to the entry and no branches out except at the
exit, is quite small.
Example: For typical MIPS programs the average branch frequency is between 15% and 25%, this means from 4 to 7 instructions execute between a pair of branches.
Data dependence can further limit the amount of ILP we can exploit within a basic block to much less than the average basic block size.
To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks (i.e. across branches).
Static detection and resolution of dependencies (static scheduling): accomplished by the compiler so dependencies are avoided by code reordering. Output of the compiler: reordered into dependency-free code. You really want a dependency free code.
Limits of Static Scheduling

- Unpredictable branches(Difficulty to determine parallelism)
- Variable memory latency (unpredictable cache misses)
- Code size explosion(Large code scaling is more costly to analyse)
- Compiler complexity