

# Advanced computer architecture fiche

Pierre Colson

June 2022

## Contents

<b>General Purpose Architectures</b>	<b>1</b>
Exploiting ILP Dynamically . . . . .	1
Pipelining . . . . .	1
Dynamic Scheduling . . . . .	2

---

Markdown version on [github](#)

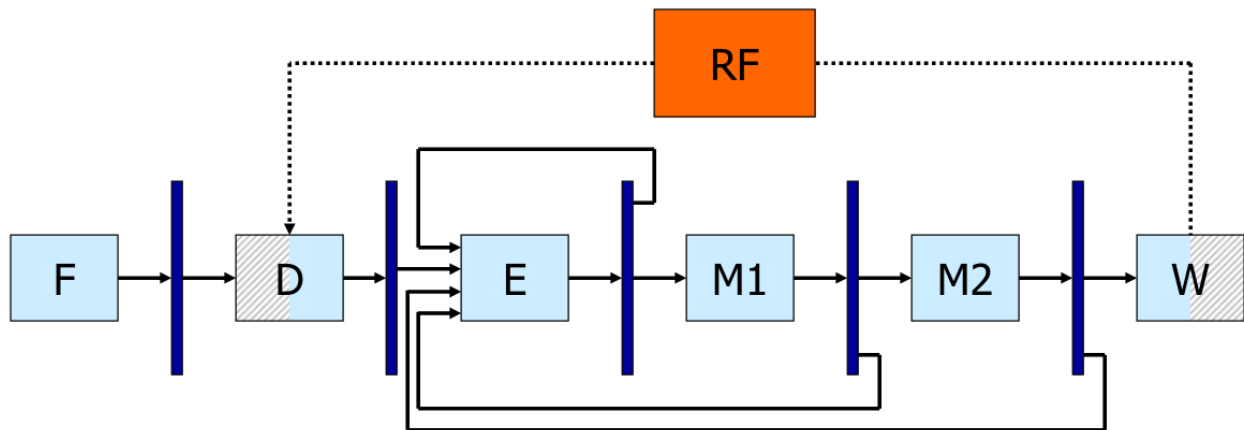
Compiled using [pandoc](#) and [gpdf script](#)

## General Purpose Architectures

### Exploiting ILP Dynamically

#### Pipelining

- **ILP** : Instruction Level Parallelism
- **Pipelining** is a simple form of ILP : Several instructions are being executed at once
  - The *latency* is unmodified
  - The *throughput* is higher
  - Adding registers to cut combinatorial logic
  - Each new register add a bit of latency and increase de power consumption
  - **Dependencies** slow down the possibility of pipelining
    - \* To detect dependencies we can simply have a look to the destination registers of intermediaire registers and verify that they are not needed by the instructions that have been fetch
    - \* Either add logic to detect dependencies (hardware) and stall the pipeline or requires the right number of nop between instructions to avoid issues (compiler)
  - Several instructions are run in parallel
  - **Control hazards** limit the usability of the pipeline
    - \* Must squash fetched and decoded instruction following a branch
  - **Data hazards** limit the usability of the pipeline
    - \* Whenever the next instruction cannot be executed, the pipeline is stalled and no new useful work is done until the problem is solved
    - \* Can be solved by forwarding newer values (bypass the register file)
  - **Rigid sequencing**
    - \* Special “slots” for everything even if sometimes useless (e.g. MEM before WB)
    - \* Every instruction must be coerced to the same framework
    - \* Structural hazards avoided “by construction”
- **Dynamic Scheduling** : solve dependencies in hardware



## Dynamic Scheduling

- While waiting for a dependency to be resolved, schedule other instruction
  - Instructions can be executed out of program order (but the result must still be correct)
- Dynamic scheduling allow **Binary compatibility**, the same code will work for different processors, Parallelism is handle in hardware on not by the compiler
- **Structural hazards**
  - Are the required resources available
  - Previously handled by rigid pipeline
- **Data hazards**
  - RAW : Are the operands ready to start execution
  - WAR and WAW : The new data overwrite something which is still required
- Dynamic pipelines may create WAW hazards
- Register after the decode state is bigger since it must be able to store several instructions. A **reservation station** checks that the operands are available (RAW) and that the execution Unit is free (Structural Hazard), then starts execution
  - Unavailable operands are identified by the name of the reservation station in charge of thr originating instruction
  - *Tag* keeps track of which instruction will generate the result I am waiting for. Tag here are the name of an entry of the reservation station
    - \* *Tag* cannot be register name since they are not unique
    - \* *Tag* cannot be the PC (even tough it looks) since we can have a branch in a loop for example
  - Can find dependencies using the reorder buffer (see later)
    - \* Reorder buffer keeps track of all instructions that have not been committed yet
    - \* Reorder buffer can be used to bypass the register file
- **Implicit register renaming** removes WAR and WAW hazards
- New results are seen at their inputs through special results bus(es)
- Writeback into the registers can be done in order or out of order
- **Architectural states** are known by the programmer
- **Microarchitectural states** are know not by the programmer, only used by the processor
- Exception handler should now exactly where a problem has occurred, especially for **non terminating exceptions** (e.g., page fault) so that they handle the event and resume exactly where the exception occurred
  - *Precise exceptions* : Reordering at commit; user view is that of a fully in-order processor

- *Imprecise exceptions* : No reordering; out-of-order completion visible to the user
- A processor can do *whatever it wants* provided that it gives the *appearance of sequential execution*
- **Reorder buffer (ROB)** : reorders instructions in the commit unit
  - Pointer to the head and to the tail
  - Every time I decode an instruction, I add it to the Reorder buffer with the corresponding *tag* and the destination register
  - During execution *tag* are replaced by actual values/results; more precisely the only instruction with the given *tag* will be updated
  - When the head pointer is ready (results has been computed) we can commit; Otherwise we wait and do not commit any instructions
  - PC is also stored in the Reoder buffer to remember where was an eventual exception. This way wen can rejump to this instruction once the exception have been handled (or not, for terminating excepetion)
  - There is an address destination slot as well as a register destination slot. Either the register or the address slot is used, because the results has to be written either to memory or to a register
- Informations/results bypass the register file in order to be reuse faster, however in register file results are in order
- When a synchronus exception happens, we do not report it but we **mark the entry** corresponding to the instruction which caused the exception in the ROB
  - When we would be ready to **commit** the instruction, we **raise the exception** instead
  - We also **squash** the content of the ROB and of all RSs
- The way to detect and resolve dependencies through memory is the same as for registers; For every load, check the ROB
  - If there is **no store to the same address** in the ROB, get the value from memory
  - If ther is a **store to the same address** in the ROB either get the value (if ready) or the tag
  - If there is a store to an **unknown address** in the ROB or if the address of the load in unknown, **wait**
- Additional memory dependencies can be solved via a **Load Queue** and a **Store Queue** (mimic the ROB). They would replace the memory reservation station
  - Load queue entries have a pointer to an entry in the store queue to know every store that where before the load
- Last improvment is Superscalar, having several fetch and decode unit as well as several alu and memory unit

