Discussion 9
Instruction level parallelism

Yutong Wang

<wangyt32023@>

4/14/2025

Iron law

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instructions}} \times \frac{\text{Time}}{\text{Cycle}}$$

Boosting performance without tuning the frequency or rewriting the program.
ILP is to executing multiple instructions in parallel by:

- having multiple datapathes running simultaneously
- utilizing datapath components that are free
- reducing stall incurred data dependencies and controls

ILP 1: Pipelining

Five stages of RISC-V datapath:

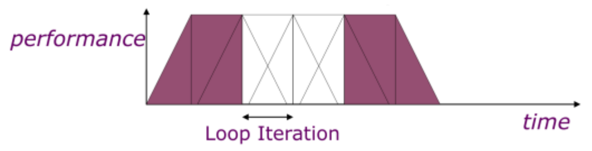
1. **IF**: instruction fetch (Read InstMem)
2. **ID**: instruction decode (Read RegFile)
3. **EX**: execution (Computation ALU)
4. **MEM**: memory access (Read/Write DataMem)
5. **WB**: write back (Write RegFile)

Time for executing one instruction cannot be shortened because of the dependencies between stages.
However, we can use inactive components to execute the next/previous instruction.

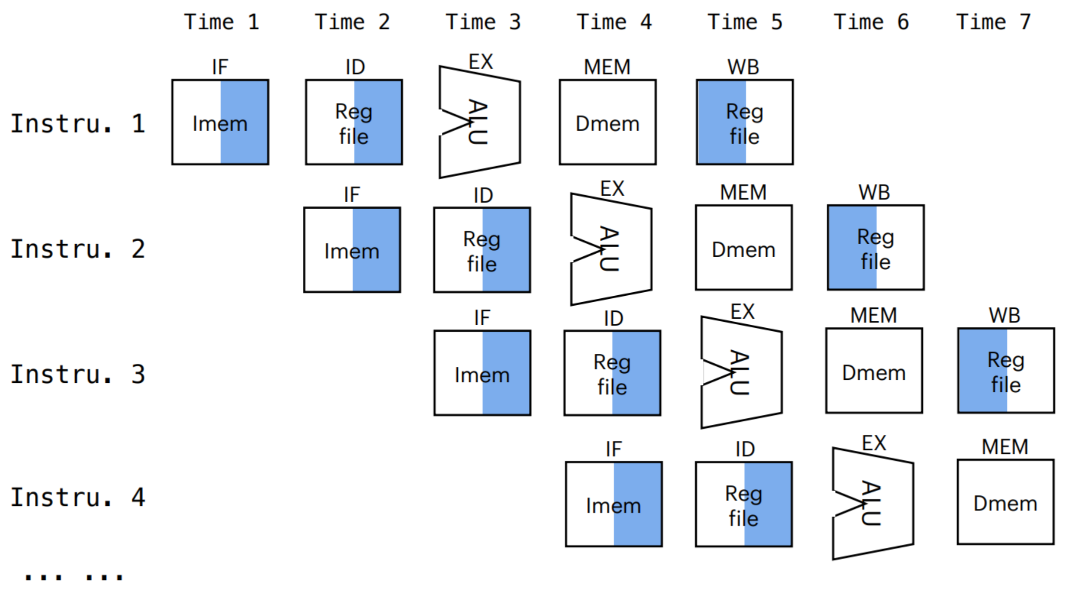
ILP 1: Pipelining

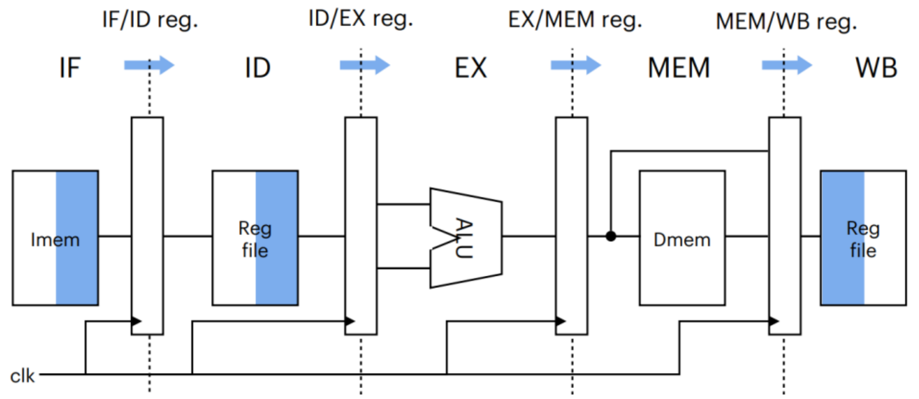
- registers between stages: prevent interference with previous/next instruction.
- Pipelining allows higher clock frequency, but typically increases the latency.

Ideal Performance: n stages $\rightarrow \mathrm{CPI} = 1, \mathrm{freq} = n \times \mathrm{freq}_{\mathrm{unpipelined}}$.



1. **Starting**: Fill pipeline stages with instructions (parallelism increase)
2. **Interim**: All stages are running simultaneously (maximum parallelism)
3. **Stopping**: Stages becomes free (parallelism decrease)





Pipeline Hazards

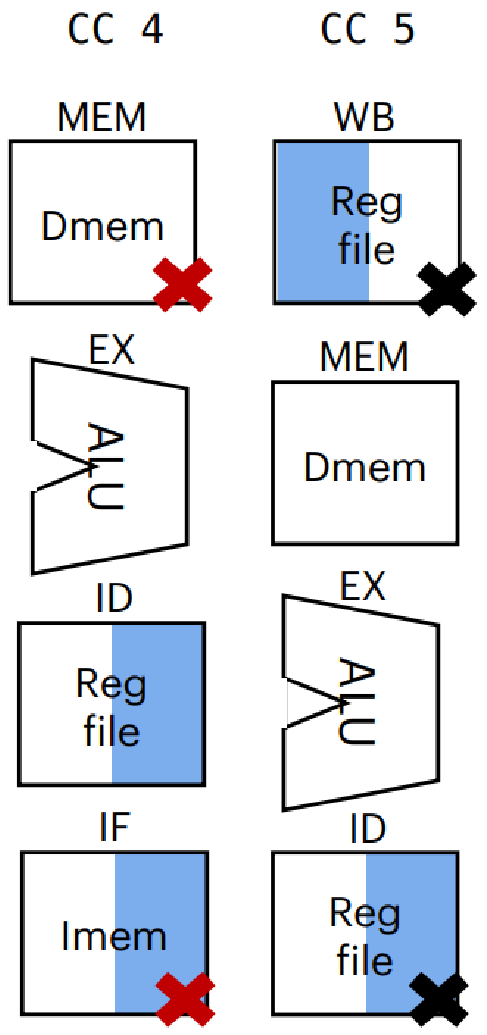
Unable to execute a stage of an instruction due to:

- Structural Hazard:
  - The required hardware resources is occupied by other instructions
- Data Hazard:
  - Dependent data not computed and stored yet
- Control Hazard:
  - Jump/Branch about to happen, unable to fetch correct instruction.

Hazard cause pipeline to stall, resulting in CPI > 1.

Structural Hazards

- On RegFile: instruction decode and register writeback
- On Mem: instruction fetch and memory read/write
- On ALU / FPU: certain computations requires more than one cycle to complete



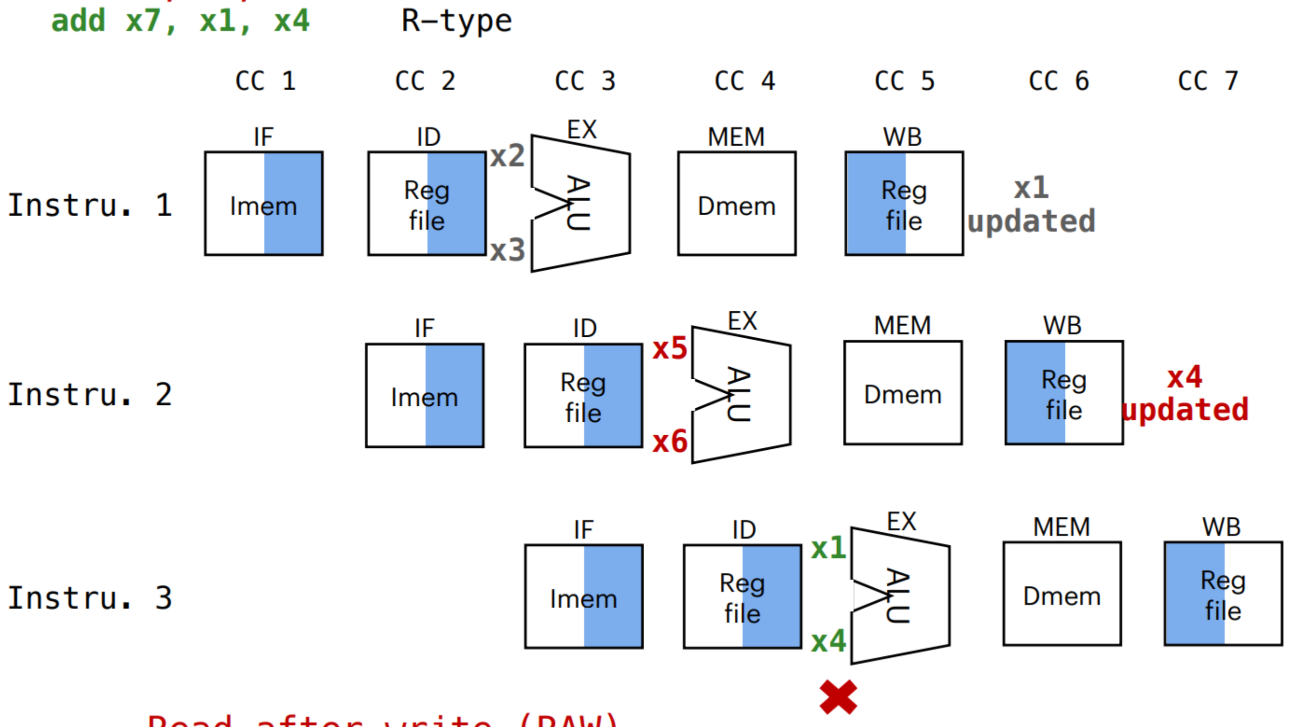
Solving structural hazards

For RegFile : W/R on rise/fall edges respectively.

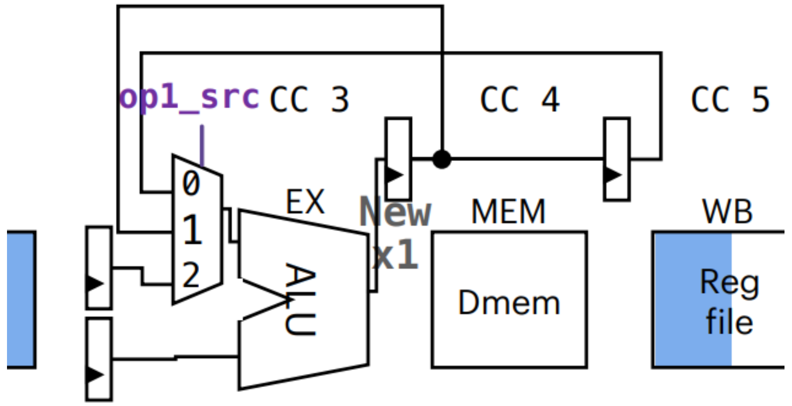
- For Mem : Separation of data memory and instruction memory.
- For ALU / FPU :
  - Re-ordering the instructions (in compile time | in runtime)
  - Adding more computation resources…

Data Hazards

```
add x1, x2, x3
add x4, x5, x6
add x7, x1, x4
```



Read after write (RAW)



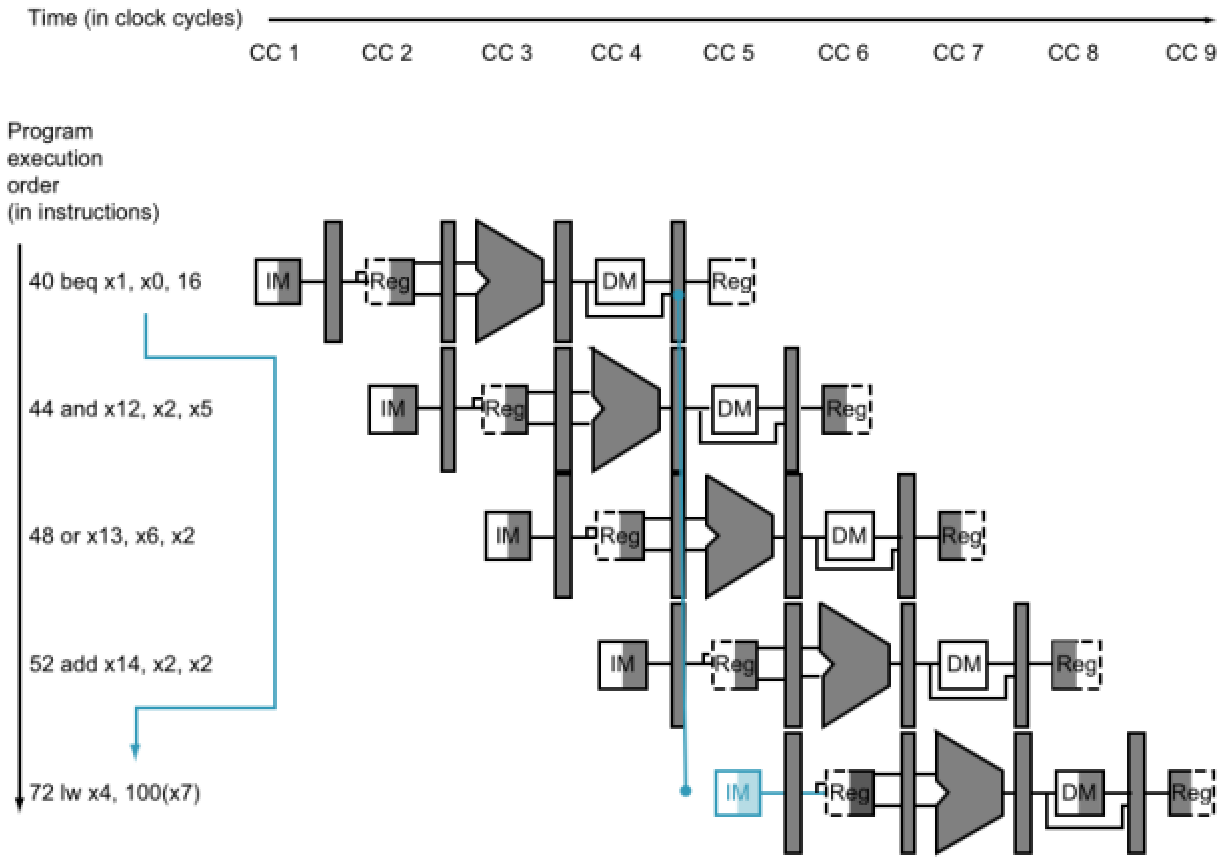
Forwarding or bypass

Unavoidable stall

- lw x1, 0(x2) : result of lw ready in cycle 4
- sub x4, x1, x5 : value of x1 used in cycle 4

Mitigation: Reorder the code, inserting an independent instruction after lw .

Control Hazards



- PC changes on rising edge of cycle 5, if branch taken.
- Unable to determine the correct instruction to fetch in cycle 2, 3, 4.

- **Idea**: continue execution, undo the wrong actions if necessary
  eager execution: executing both if branch and else branch
- **branch prediction**: guess the result of branch comparison
  - Nothing wrong on correct prediction
  - Flush pipeline on wrong prediction

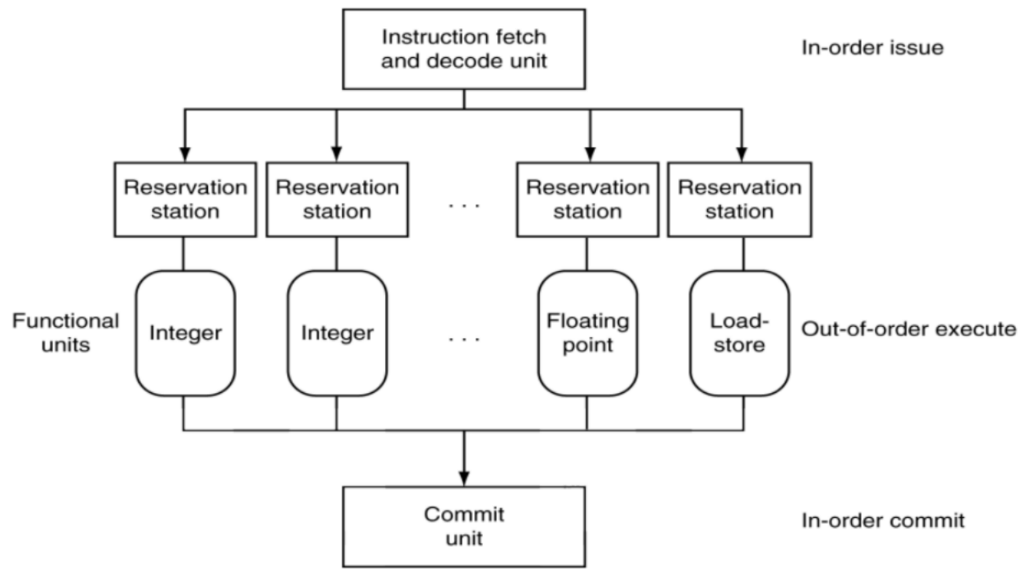
ILP 2: Multi-issue

What if we pack multiple ALUs, FPUs and Load/Store units into the processor?

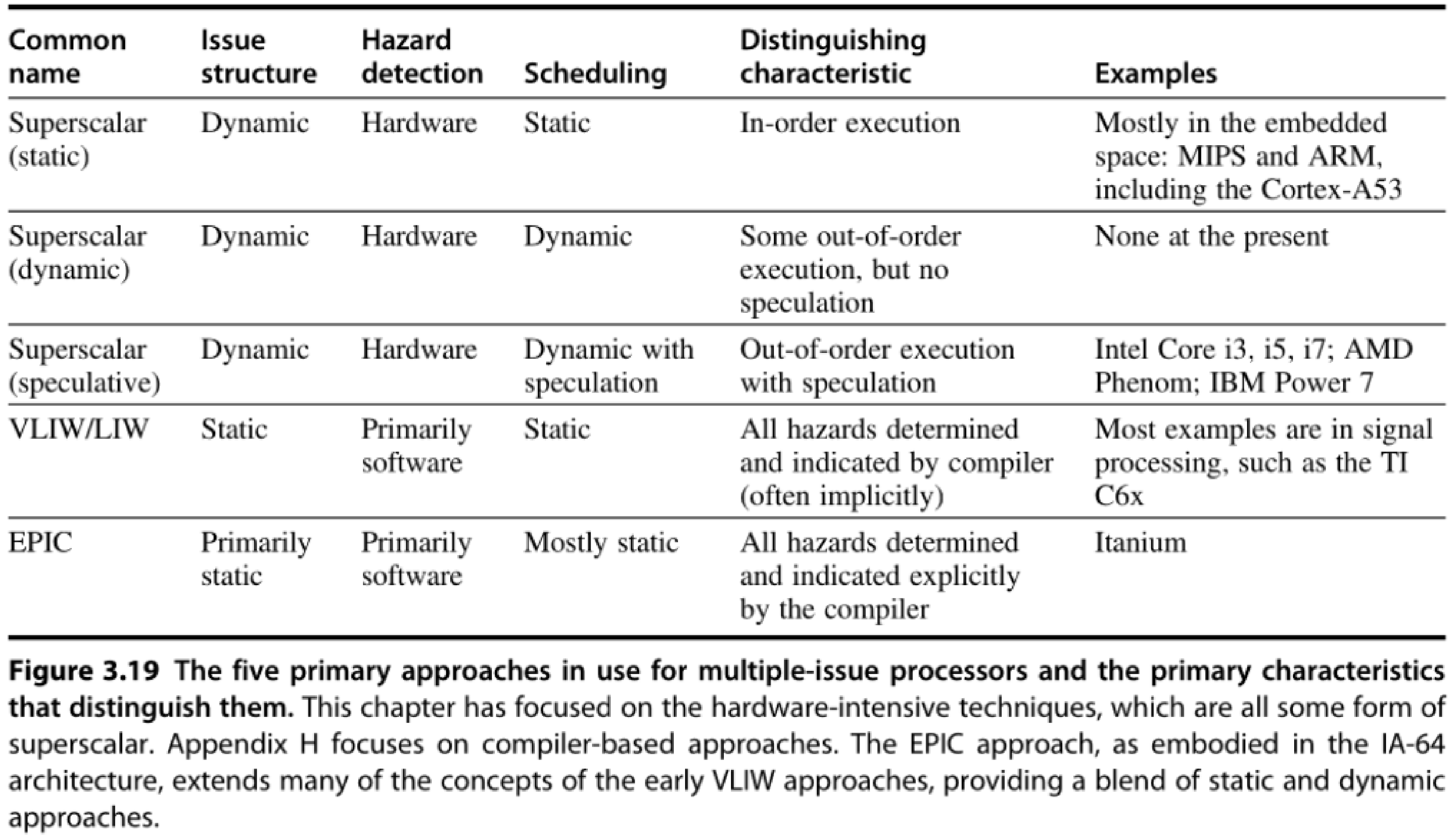
- **Single Issue**: fetching and start executing a single instruction per cycle. Not utilizing the extra computation power.
  - Ideal performance: CPI = 1
- **Multiple Issue**: fetching and starting executing multiple instructions per cycle. Exploiting the additional hardware resources.
  - Ideal performance: CPI < 1

Typically paired with the following techniques:

- Out of order execution: help mitigating data dependencies (RAW, WAR, WAW)
- Speculative execution: help reduce impact of branches



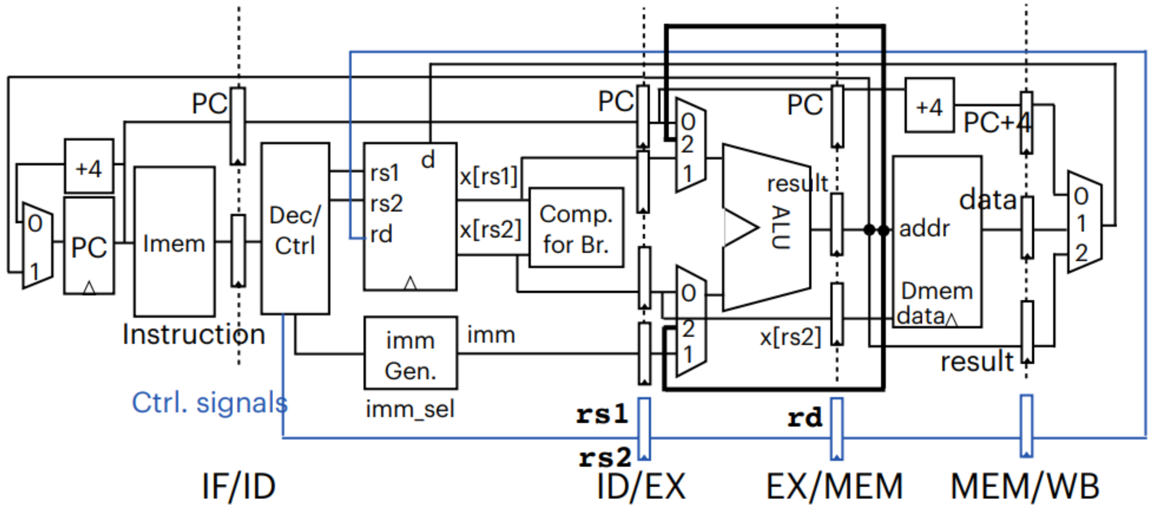
Appendix



Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Figure 3.19 The five primary approaches in use for multiple-issue processors and the primary characteristics that distinguish them. This chapter has focused on the hardware-intensive techniques, which are all some form of superscalar. Appendix H focuses on compiler-based approaches. The EPIC approach, as embodied in the IA-64 architecture, extends many of the concepts of the early VLIW approaches, providing a blend of static and dynamic approaches.

Exercise



The given code is executed on the processor:

```
ADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, R7
OR  R8, R6, R9
```

CC	1	2	3	4	5	6	7	8	9	10
ADD										
SUB										
AND										
OR										