Discussion 9 : Instruction level parallelism in CA-I

Contents

- Motivations
- Techniques exploting ILP
  - Pipelining
  - Static Multiple Issue: VLIW
  - Dynamic Multiple Issue: Superscalar
  - Speculative Execution
  - Out-of-Order Execution
- Exercises

Why ILP

Recall: The iron law of performance

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

Boosting performance without tuning the frequency or rewriting the program.

ILP is to executing multiple instructions in parallel by:

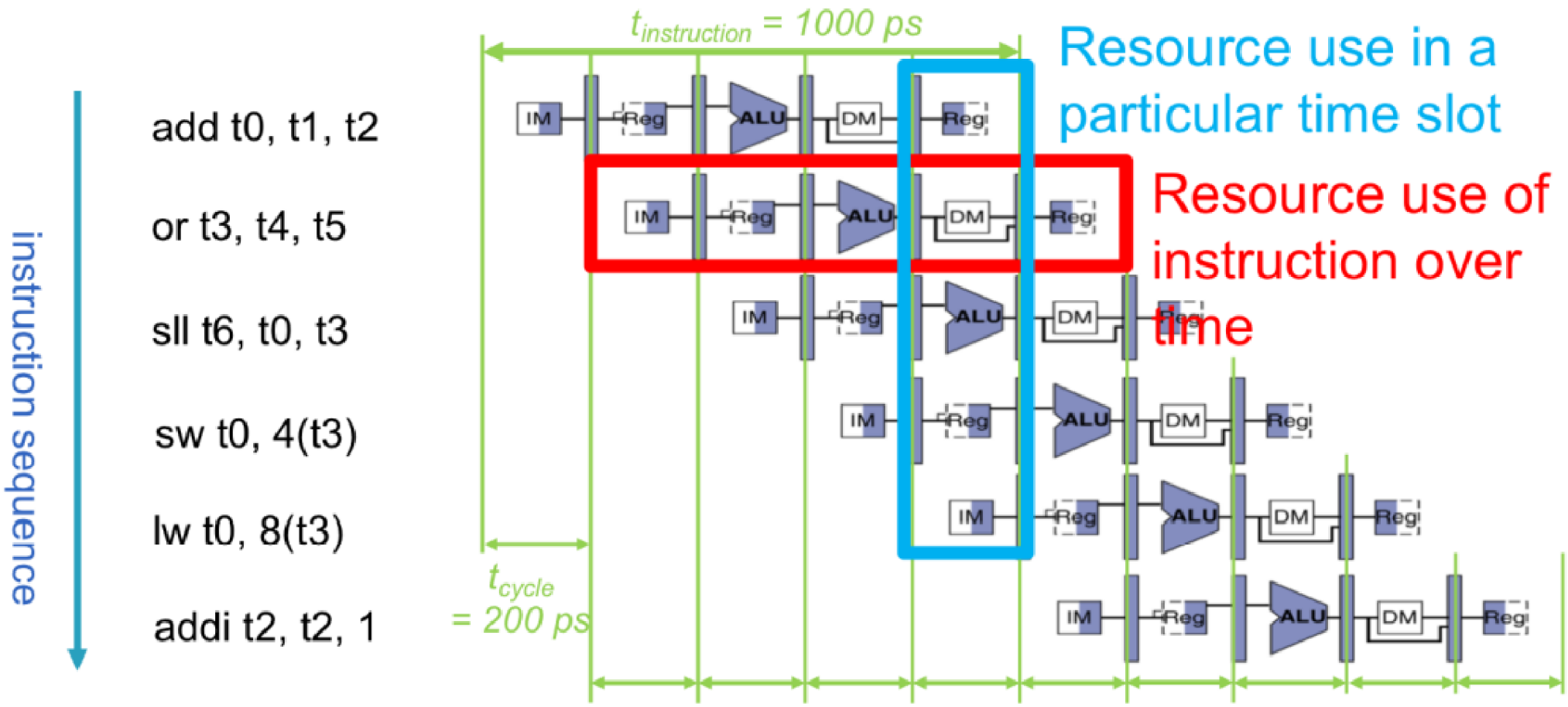
- having multiple datapathes running simultaneously
- utilizing datapath components that are free
- reducing stall incurred data dependencies and controls

Exploting ILP 1: Pipelining

Recall: 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.



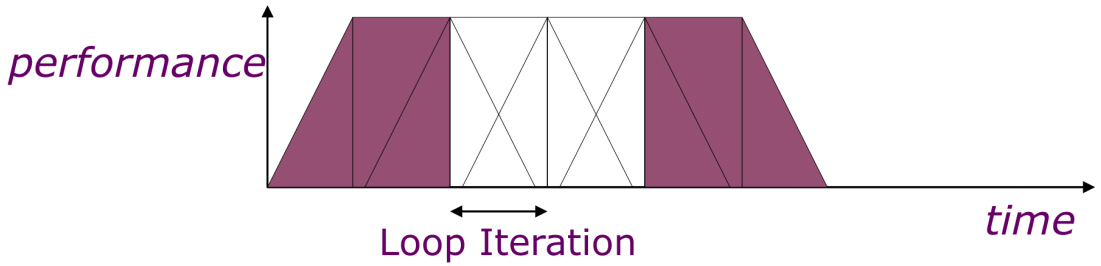
note:

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

Pipeline: Performance

- without pipeline: One of (IMEM, RegFile, ALU, DMEM) is in use.
- pipelined process: All stages are running, all of (IMEM, RegFile, ALU, DMEM) are in use.

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 n stages are running simultaneously (maximum parallelism)
3. Stopping: Stages becomes free (parallelsim decrease)

Pipelining: Hazards

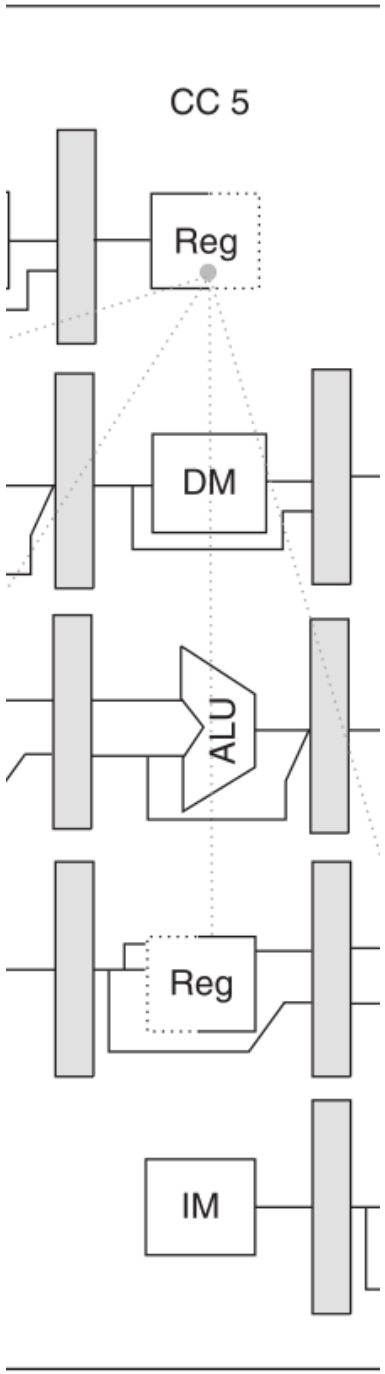
Unable to execute a stage of an instruction due to:

- **Strutrual 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 $\mathrm{CPI} > 1$

Structural hazards

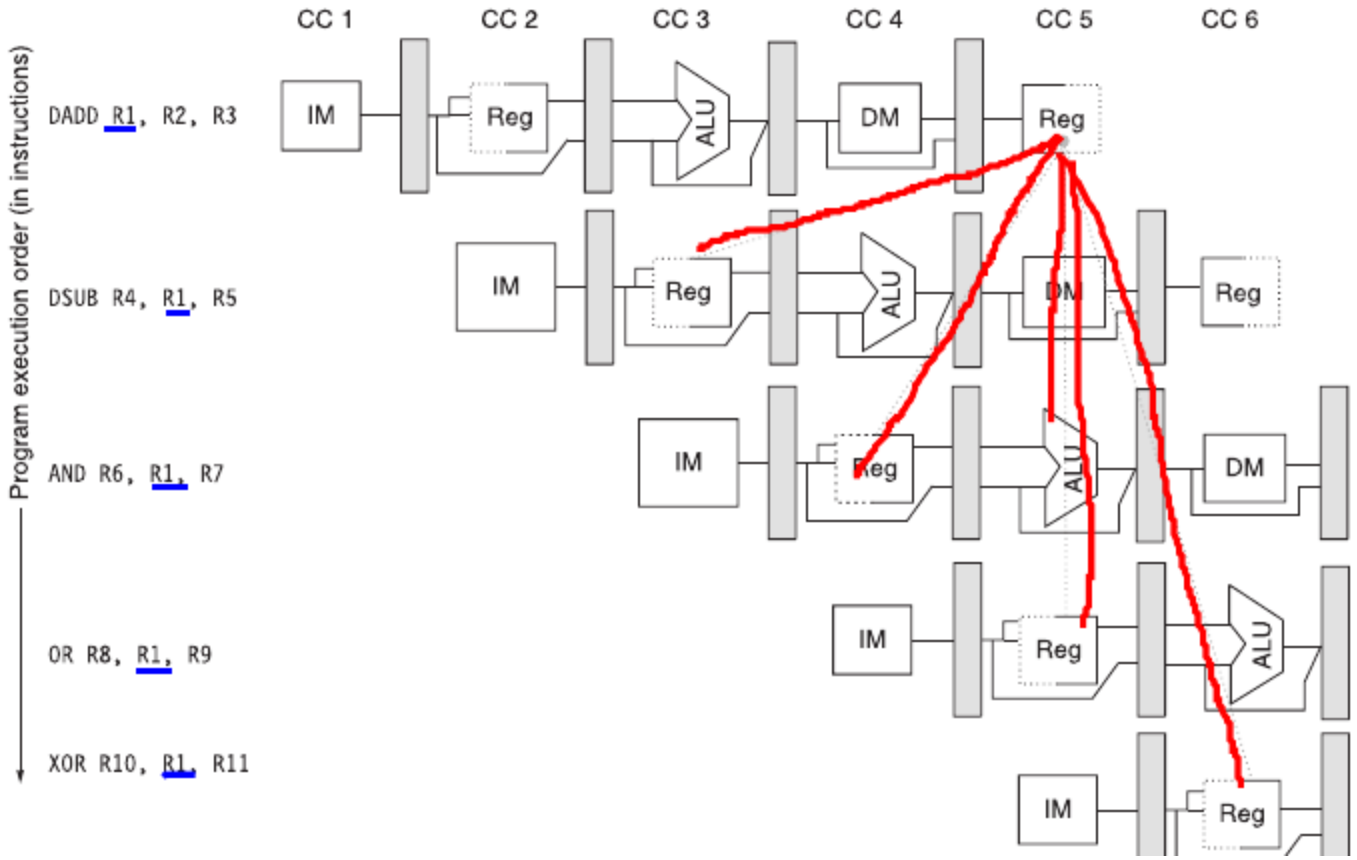
- On `RegFile`
  instruction decode and
  register writeback
- On `Mem`
  instruction fetch and memory
  read/write
- On `ALU` / `FPU`
  certain computations requires
  more than one cycle to
  complete



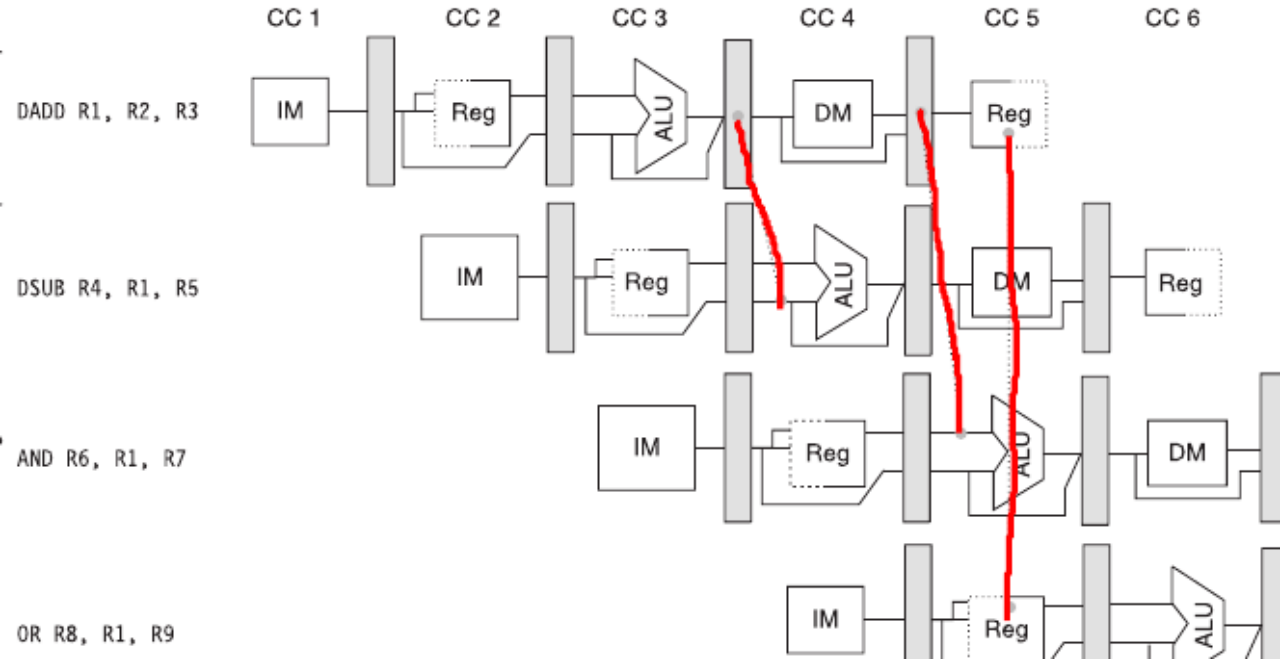
Solving strutral hazards

- 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



We mainly concern the read after write (RAW) dependency.

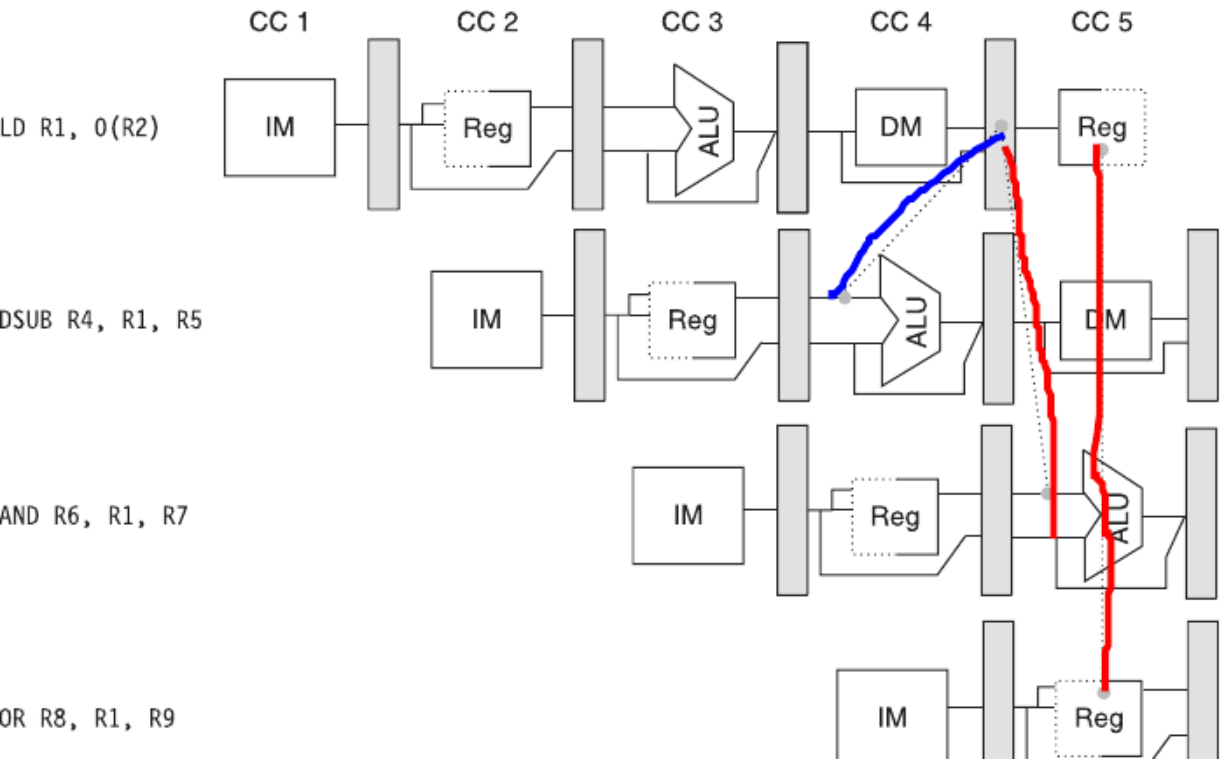


Resolving RAW with forward

Addition result is ready in cylce 3.

1. `sub` : result used in cycle 4
2. `and` : result used in cycle 5
3. `or` : result fetched in cycle 6

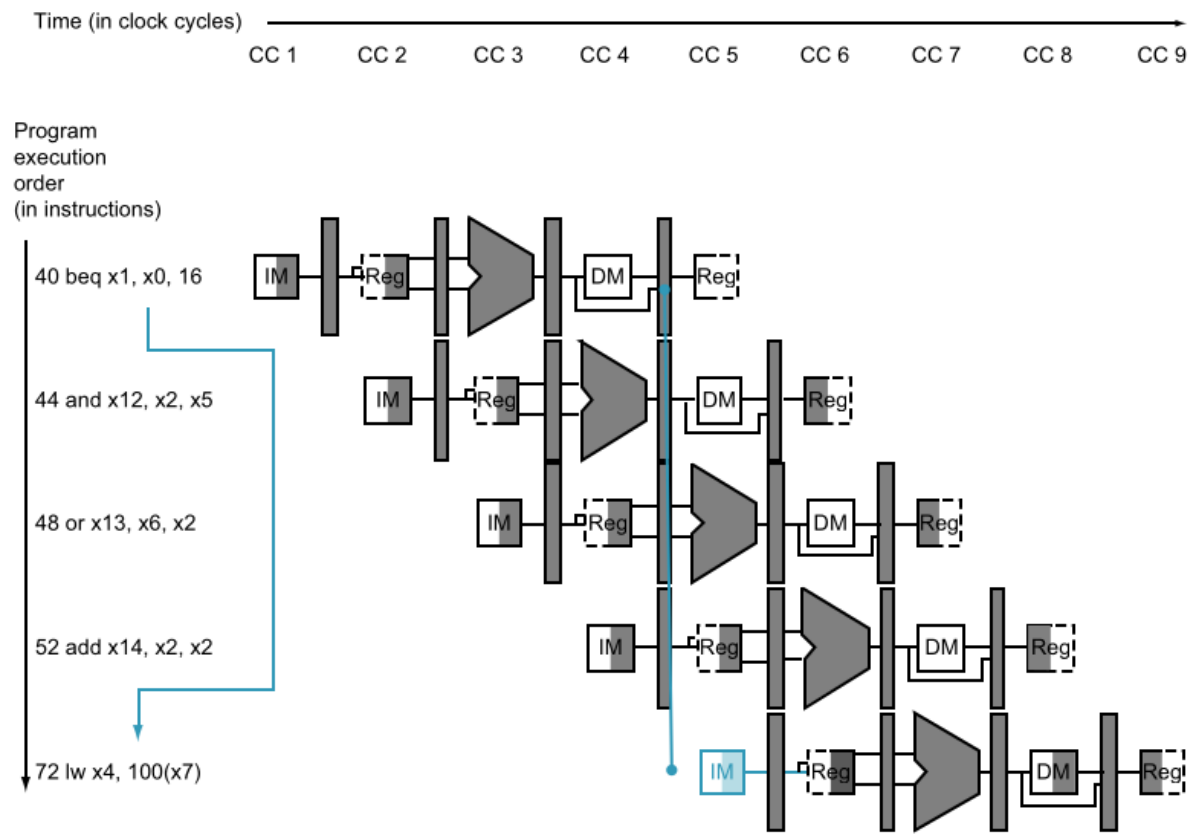
Replace the out-dated value from register with the result from ALU.



Unavoidable stall

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

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



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.

Speculative execution

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
    (branch compare result available in cycle 3, 2 instructions affected)

Performance tip for C/C++ programmer:

- short-circuit logical operators `&&` and `||` involves branch perdiction.
- bitwise operators `&&` and `|` are egarly evaluated.
- See also: CppCon 2021 branchless programming video slides

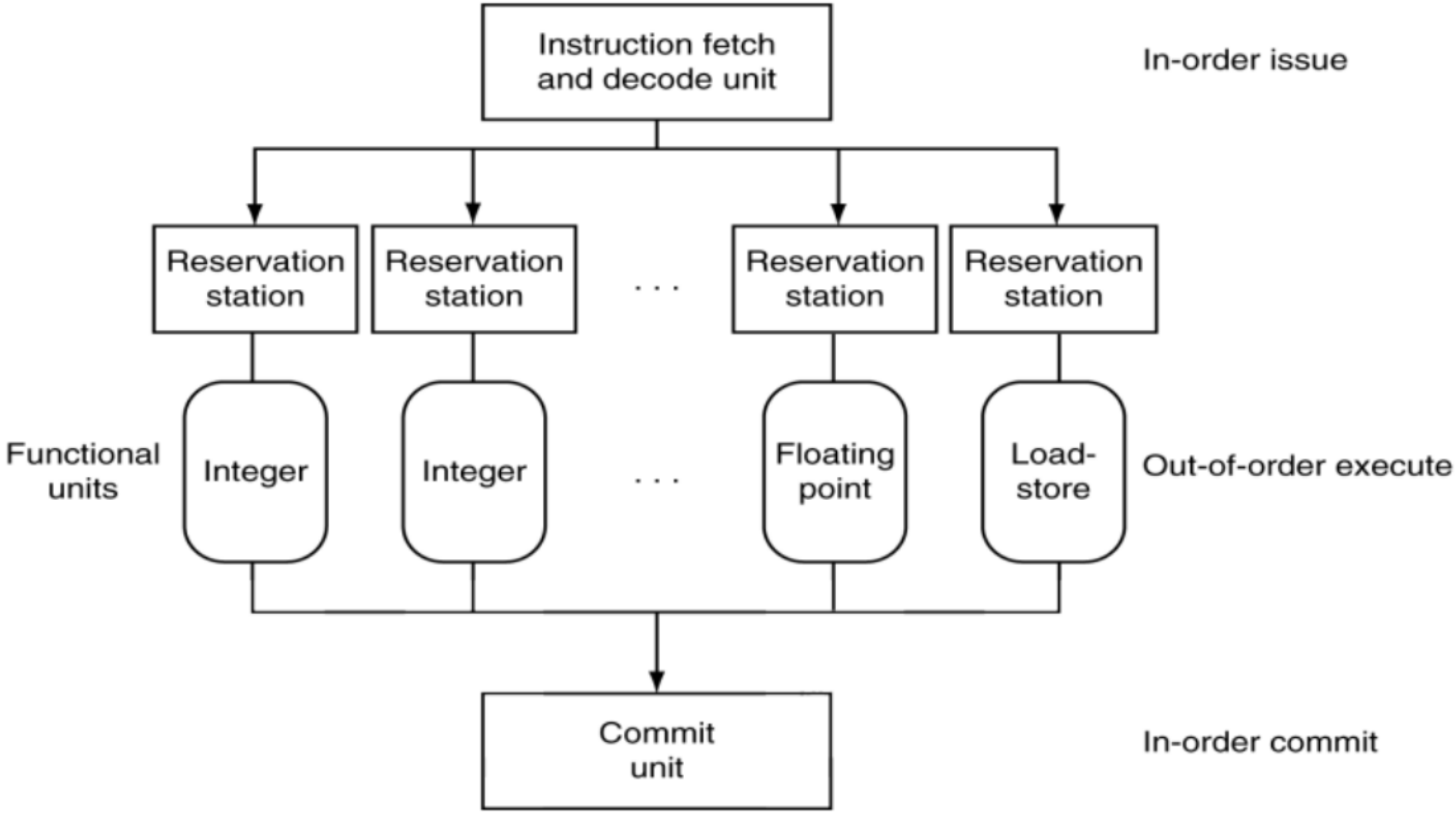
Exploting ILP 2: Multiple Issue

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: $\mathrm{CPI} = 1$
- Multiple Issue: fetching and starting executing **multiple** instructions per cycle.
  Exploting the additional hardware resources.
  Ideal performance: $\mathrm{CPI} < 1$

Typically paired with the following techniques:

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

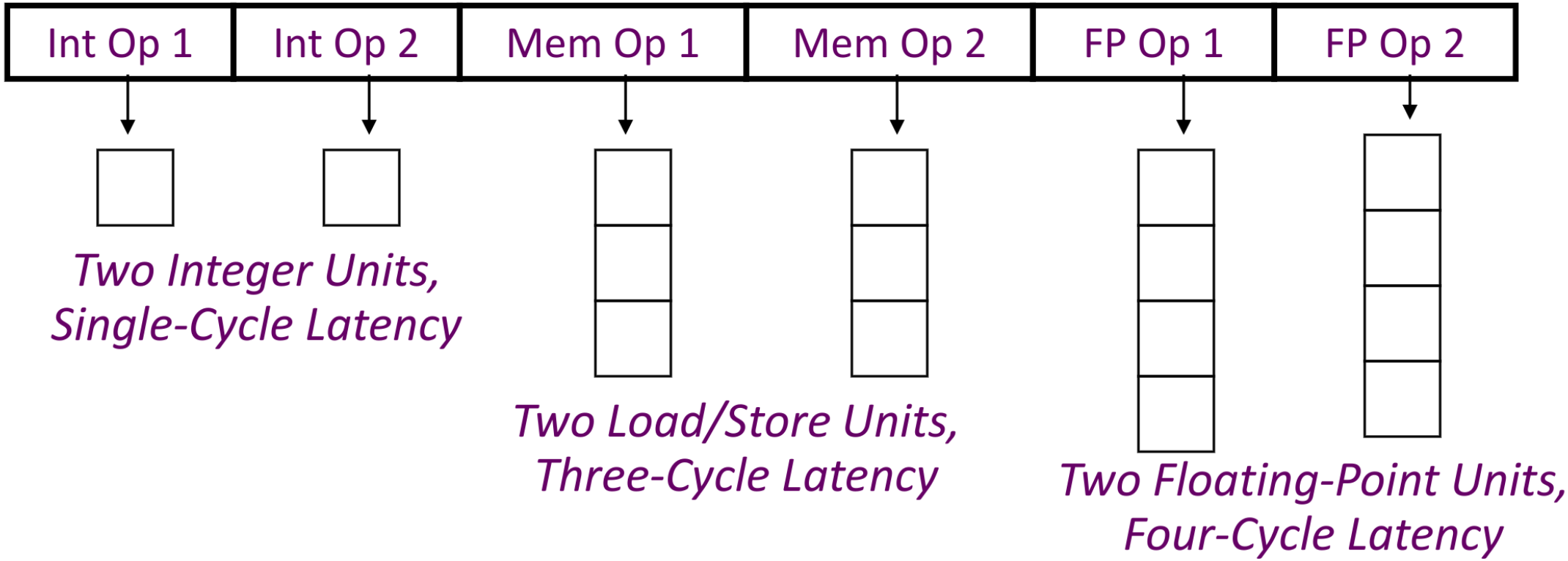

Instruction fetch and decode unit
In-order issue
Reservation station
Reservation station
. . .
Reservation station
Reservation station
Functional units
Integer
Integer
. . .
Floating point
Load-store
Out-of-order execute
Commit unit
In-order commit

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.

VLIW: Very Long Instruction Word

Instead of using a simply 32-bit instruction word to encode one operation, VLIW packs multiple operators into one instruction word (generally longer).



Static issuing and static scheduling.

Compiler responsible for identifying possible parallelism and grouping OPs

Superscalar

To fetch, issue for execution and finish more than one instruction per cycle.

t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8
F1	F3	F5	F7	F9				
F2	F4	F6	F8	F10				
	D1	D3	D5	D7	D9			
	D2	D4	D6	D8	D10			
		X1	X3	X5	X7	X9		
		X2	X4	X6	X8	X10		
			M1	M3	M5	M7	M9	
			M2	M4	M6	M8	M10	
				W1	W3	W5	W7	W9
				W2	W4	W6	W8	W10

Example of a 2-issue, 5-stage pipelined prokkcessor executing 10 instructions.

Hardware responsible for dependencies analysis and dynamic issuing.

Additional Exercises

- previous exams of CS110@ShanghaiTech
- previous exams of CS61c@Berkeley
- previous exams of CS211@ShanghaiTech
- previous exams of CS152@Berkeley

Reference & Acknowledgement

- Slides from **CS211@ShanghaiTech** by Prof. Chundong Wang
- Slides from **CS152@UC Berkeley** by Krste Asanovic
- Chapter 4 of **Computer Organization and Design 2nd edition**
- Chapter 3 of **Computer Architecture: A Quantitative Approach**