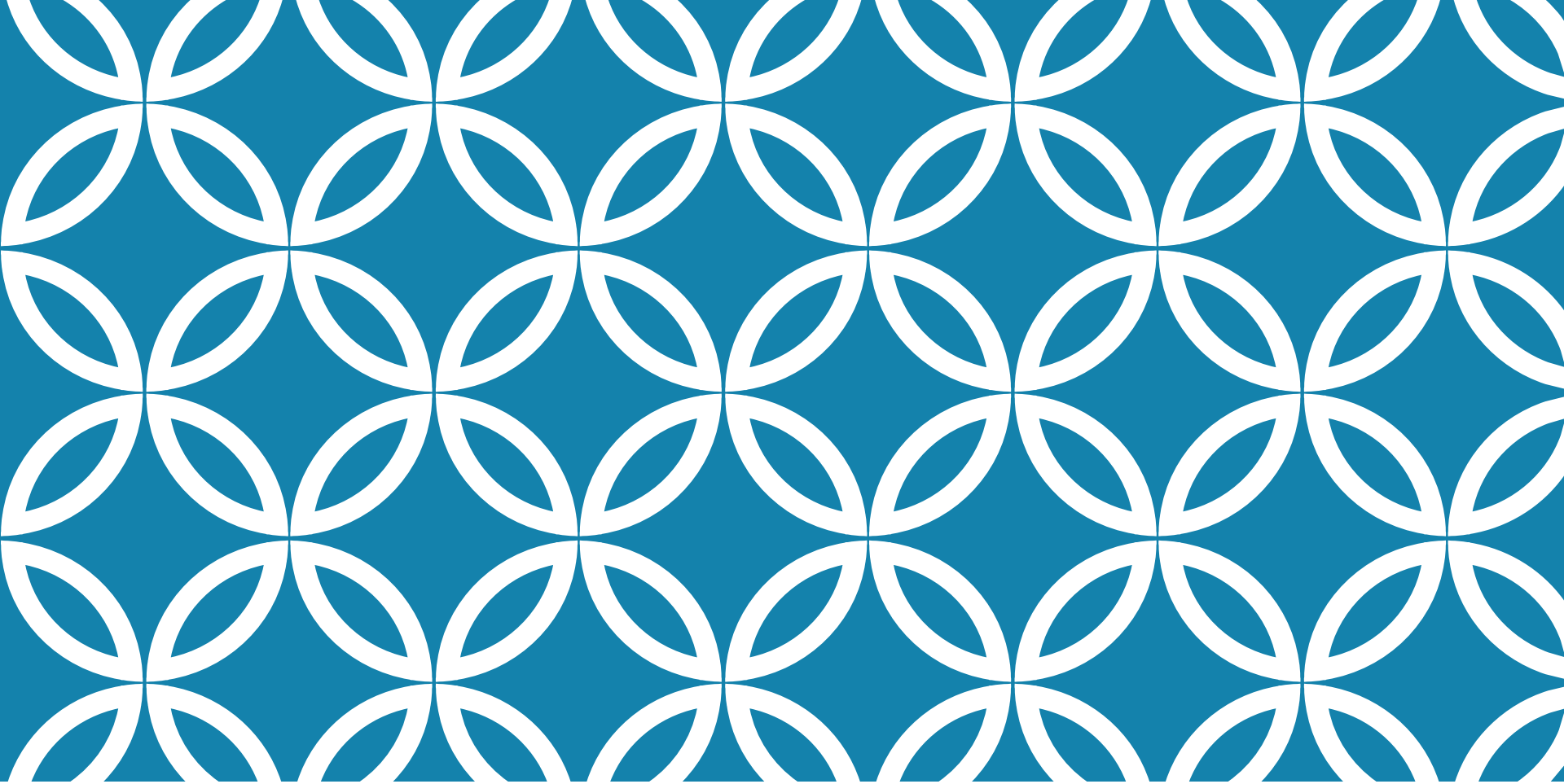


CHAP-16; INSTRUCTION LEVEL PARALLELISM AND SUPER SCALAR PROCESSOR

Computer Organization and Architecture Designing for Performance. 9th Edition. William Stallings.

Issues with out of order completion

- Output dependency (WAW hazard)
- Anti dependency (WAR hazard)

Output dependency

- When two instructions happen to be having same destination and second instruction try to write before first instruction, for example

$I1$: $MUL\ R1, R2, R3,$
I2: $ADD\ R1, R4, R5,$

Or

I1: $R3 \leftarrow R3\ OP\ R5$; OP is any operation

I2: $R4 \leftarrow R3 + 1$

I3: $R3 \leftarrow R5 + 1$

I4: $R7 \leftarrow R3\ OP\ R4$

— True dependency exists between I1,I2 and I3,I4 (RAW hazard)

What about I1,I3?

— Wrong value of I3 is used by I2, if I3 writes before I1 does.

— Usually known as WAW hazard

— Solution?

Anti-dependency

- An anti-dependency exists if an instruction uses a location as an operand while a second one is writing into that location, for example

$$I1\colon\ MUL\ R4, R3, R1$$
$$I2\colon\ ADD\ R3, R2, R5$$

— **Problem arises if second instruction updates R3 before first instruction uses it.**
— **Usually known as WAR hazard**
— **Often appears in out of order completion processors.**

— **Solution?**

Output and Anti-dependencies (Cont'd)

- Output dependencies and Anti dependencies aren't true dependencies.
- Second instruction doesn't need operands calculated by prior instructions.
- These dependencies arise due to limited number of registers available.
- These dependencies can be solved by using a technique called Register Renaming
- Processor detects these hazards and rename those dependent registers.

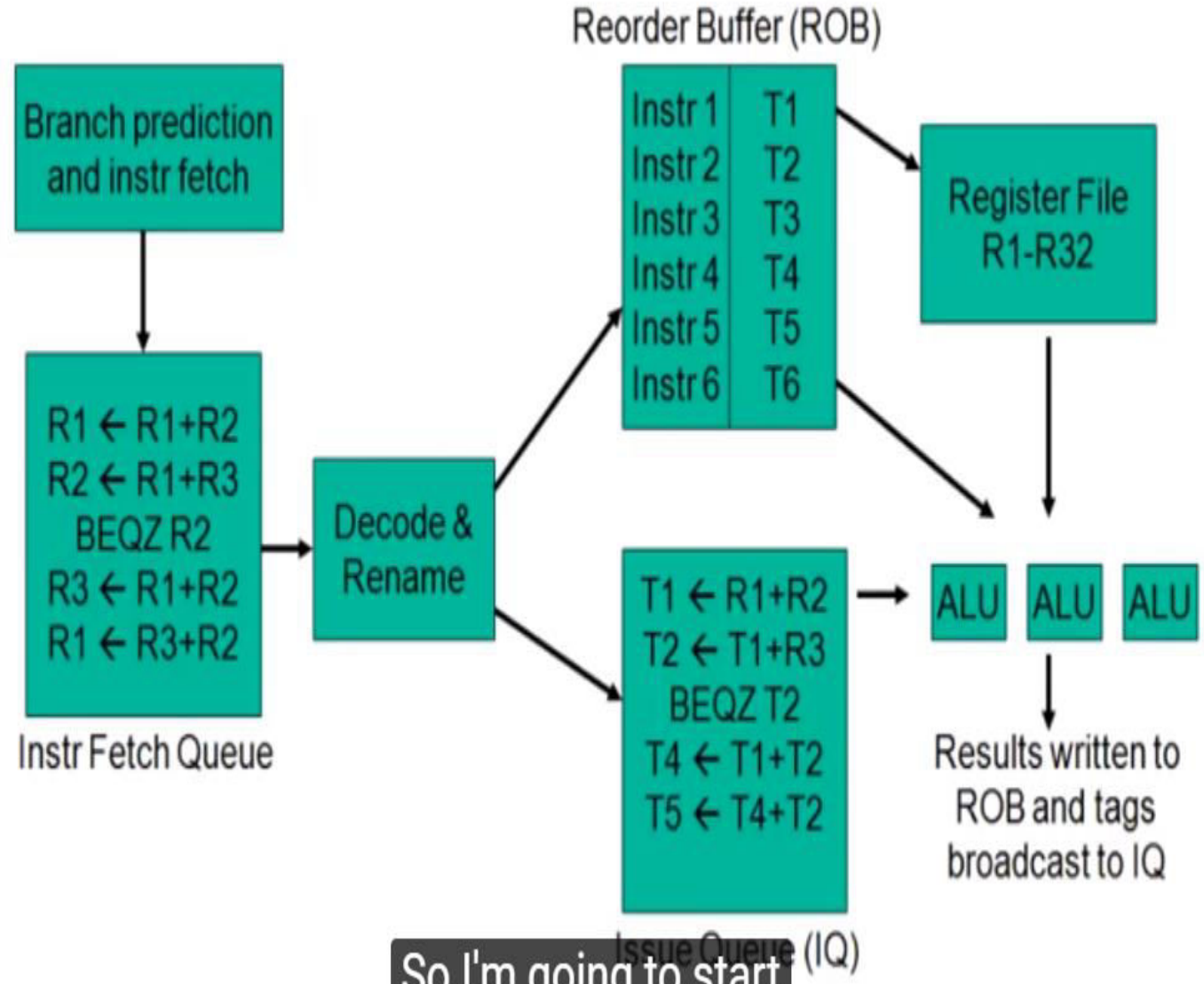
Output and Anti-dependencies (Cont'd)

- I1: $R3 \leftarrow R3\ OP\ R5$
  I2: $R4 \leftarrow R3 + 1$
  I3: $R3 \leftarrow R5 + 1$
  I4: $R7 \leftarrow R3\ OP\ R4$

  I1,I3 show Output dependency (WAW hazard, same destination operand).
  I2,I3 show Anti dependency (WAR hazard, I3 can write before I2 reads operands)

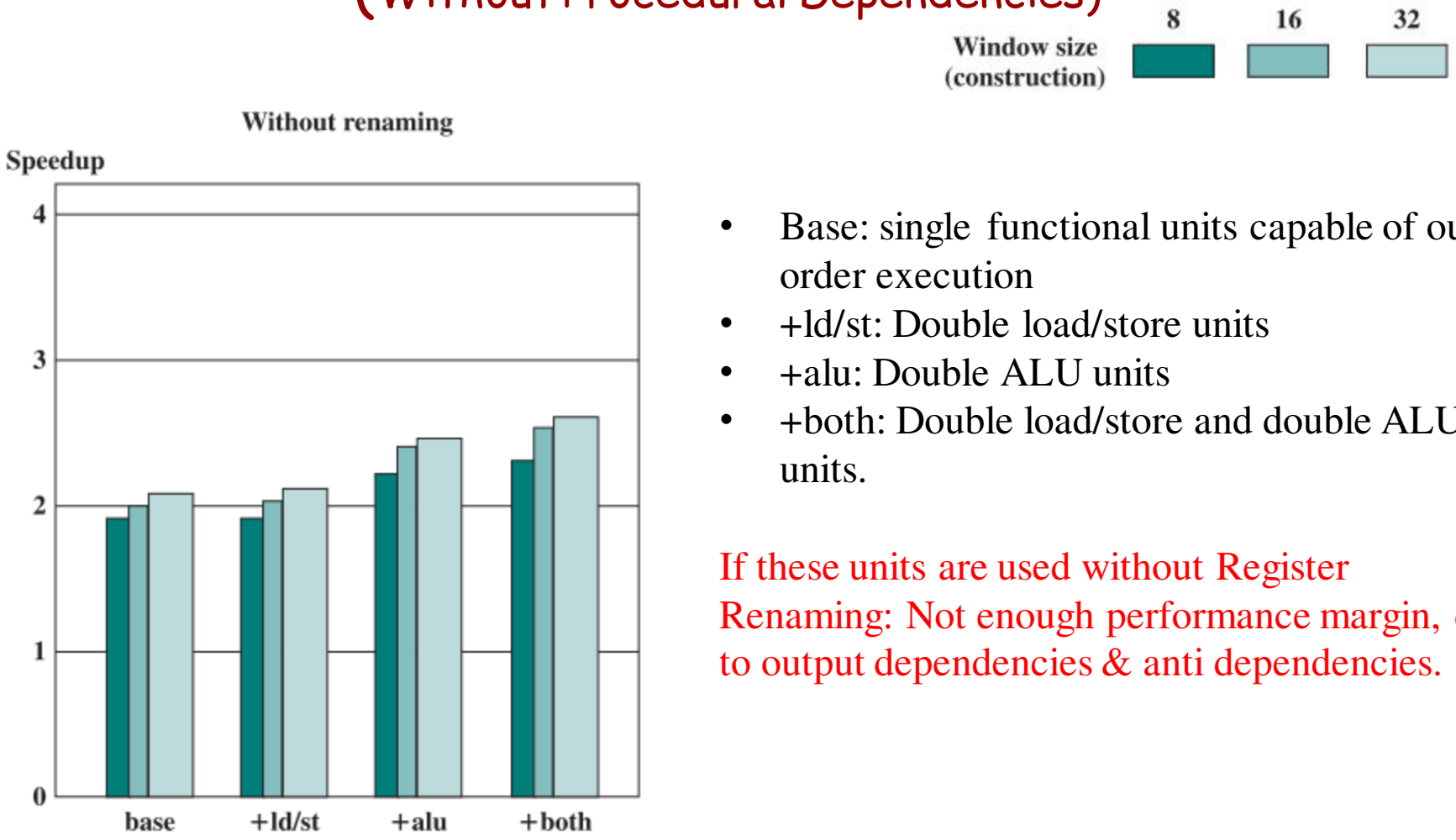
  I1: $R3_b \leftarrow R3_a\ OP\ R5_a$
  I2: $R4_b \leftarrow R3_b + 1$
  I3: $R3_c \leftarrow R5_a + 1$
  I4: $R7_b \leftarrow R3_c\ OP\ R4_b$

An Out-of-Order Processor Implementation



So I'm going to start with this figure

Speedups of Machine Organizations
(Without Procedural Dependencies)



- Base: single functional units capable of out of order execution
- +ld/st: Double load/store units
- +alu: Double ALU units
- +both: Double load/store and double ALU units.

If these units are used without Register Renaming: Not enough performance margin, due to output dependencies & anti dependencies.

Figure 16.6 Speedups of Various Machine Organizations without Procedural Dependencies

Speedups of Machine Organizations
(Without Procedural Dependencies)

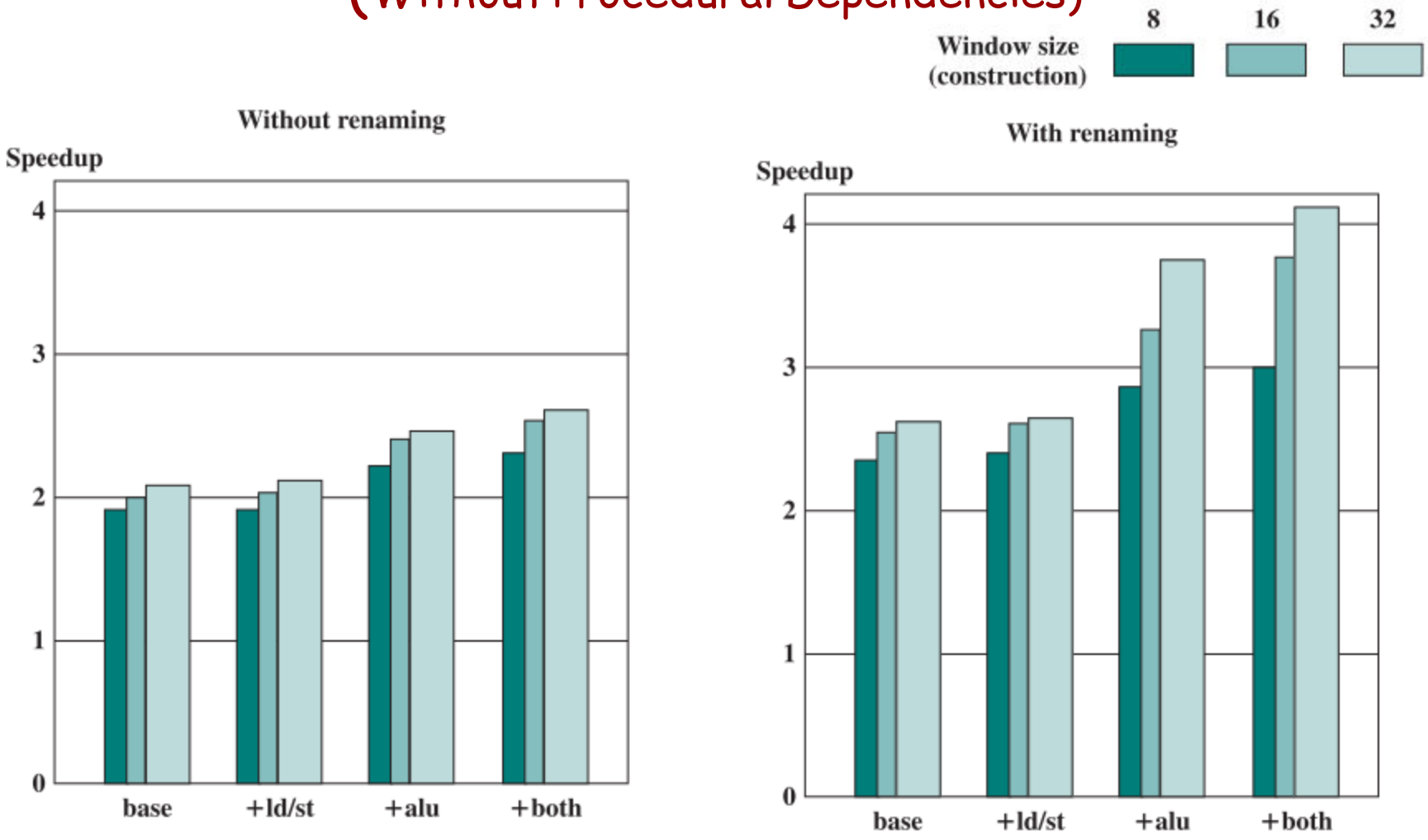


Figure 16.6 Speedups of Various Machine Organizations without Procedural Dependencies

- Performance improves considerably, if Register Renaming used.
- Need instruction window large enough (more than 8, probably not more than 32)

Committing or Retiring Instructions

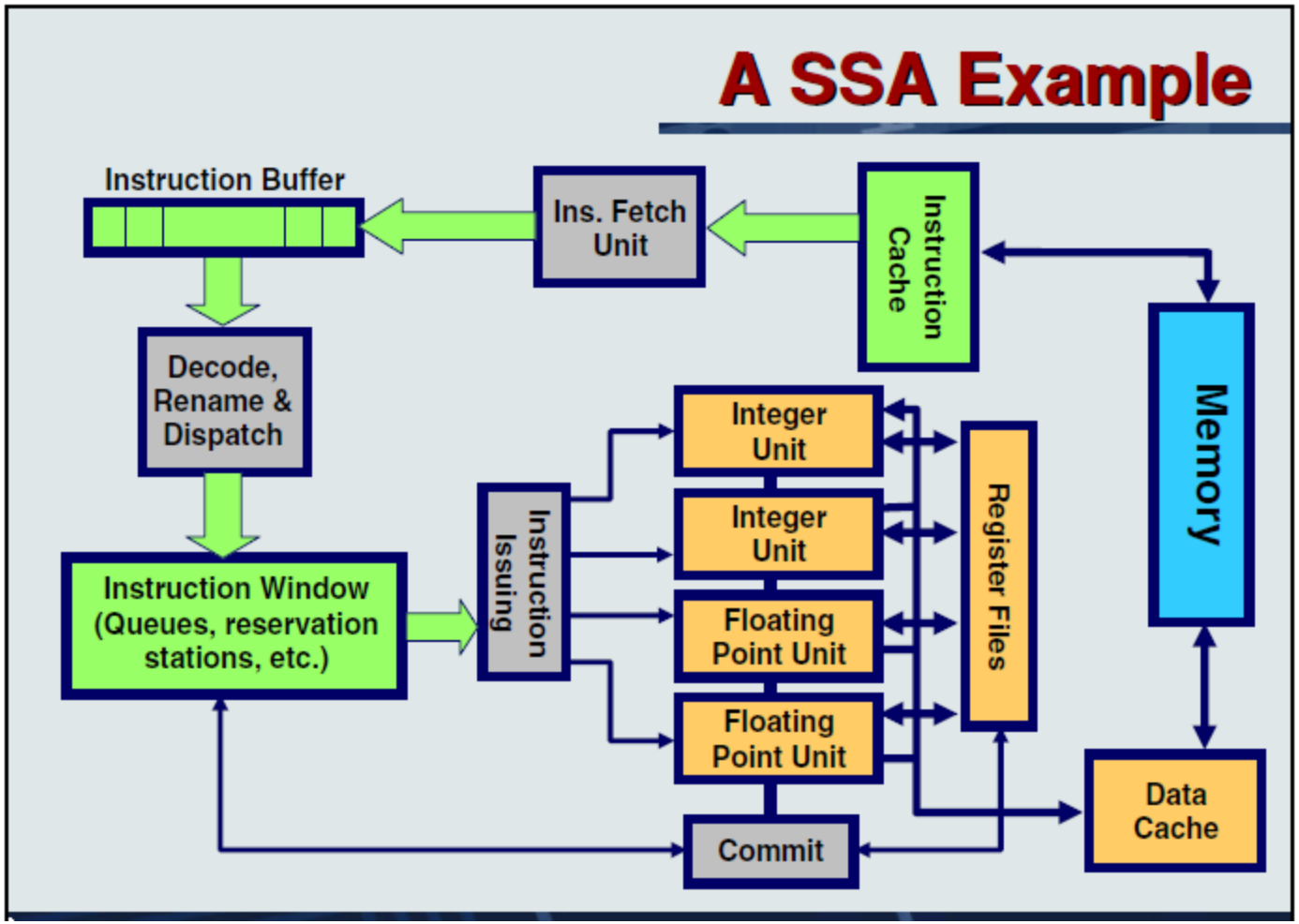
Results need to be put into order (commit or retire)

- Results sometimes must be held in temporary storage until it is certain they can be placed in "permanent" storage.

  (either committed or retired/flushed as in case of branch instructions)

- Temporary storage requires regular clean up – overhead – done in hardware.

A SSA Example

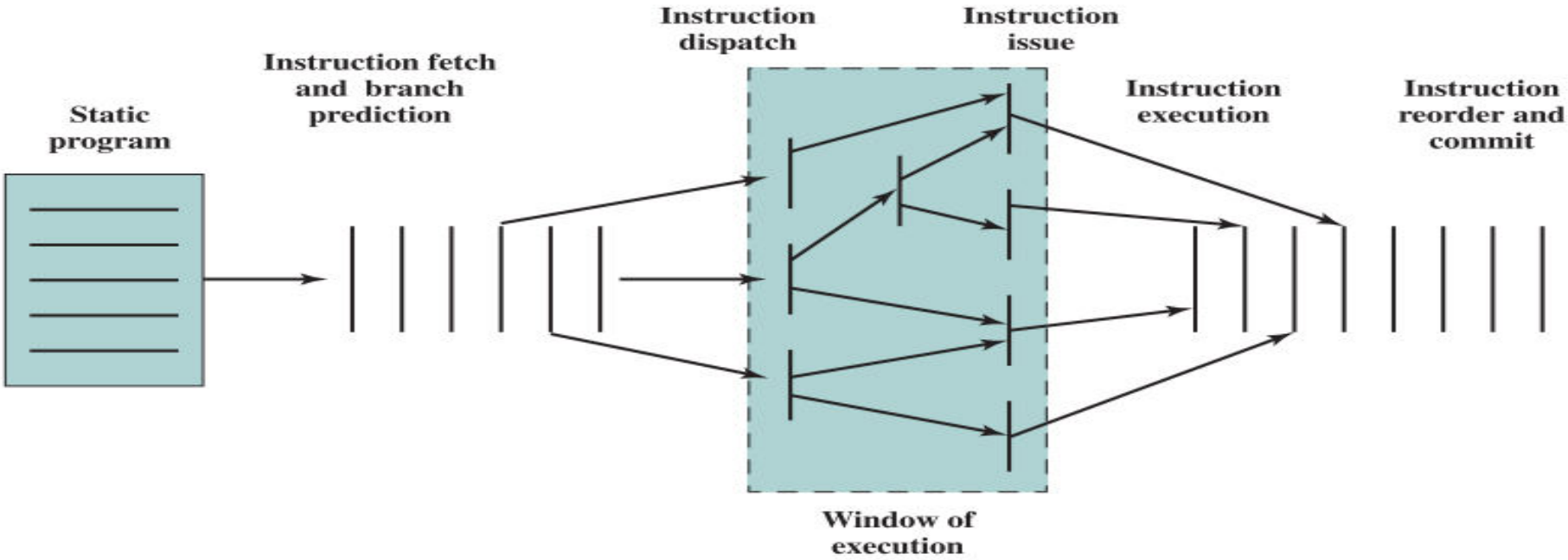


Superscalar Processing

- Fetch (supply N instructions)
- Decode (generate control signals for N instructions)
- Rename (detect dependencies between N instructions)
- Dispatch (determine readiness and select N instructions to execute in-order or out-of-order)
- Execute (have enough functional units to execute N instructions + forwarding paths to forward results of N instructions)
- Write into Register File (have enough ports to write results of N instructions)
- Retire (N instructions)

View of Superscalar Execution

- Static Program are instructions, written by the programmer or generated by the compiler.
- The instruction fetch stage includes branch prediction which is used to form a dynamic stream of instructions.
- This stream is examined for dependencies, and the processor may remove artificial dependencies by hardware techniques (Stall, Forward, Register Renaming)
- The processor then dispatches the instructions into a window of execution (Instruction window)
- In this window, instructions are structured according to their true data dependencies.
- The processor executes each instruction in an order determined by the true data dependencies and hardware resource availability.
- Finally, instructions are conceptually put back into sequential order and their results are recorded.
- And in final stage, instructions are committed or retired.



Recaping: Machine Parallelism Support

- Duplication of Resources (Execution units, memories, register file)
- Out of order issue hardware (sorts out independent instructions)
- Decouple execution stage from decode stage and introduce Instruction window (Buffer) to hold decoded instructions.

Superscalar Hardware Support

- Facilities to simultaneously fetch multiple instructions
- Logic to determine true dependencies involving register values and Mechanisms to communicate these values
- Mechanisms to initiate multiple instructions in parallel
- Resources for parallel execution of multiple instructions
- Mechanisms for committing process state in correct order

Problem 1

Consider the following assembly language program:
I1: R3 $\leftarrow$ R3 op R5
I2: R4 $\leftarrow$ R3 + 1
I3: R3 $\leftarrow$ R7 + 1
I4: R7 $\leftarrow$ R3 op R4
I5: R5 $\leftarrow$ R4 + 3
I6: R4 $\leftarrow$ R3 op R6
I7: R7 $\leftarrow$ R4 + 4

(a) Write RAW, WAW, and WAR dependencies in the above code segment.

(b) Remove WAW and WAR using register renaming.

(c) Consider a super scalar with the following properties:

(i) It can decode two instructions at a time

(ii) It has 2 functional units. Both functional units can perform op and +. op takes two clock cycles to execute and + takes one clock cycles to execute.

(iii) It can write two instructions at a time.

Write down all the constraints for the execution of the given code on the super scalar. Assume WAW and WAR dependencies are solved using register renaming.

(d) Compute the clock cycles to execute the program on the given super scalar machine using:

(a) In-order issue with in-order completion

(b) In-order issue with out-of-order completion

(c) Out-of-order issue with out-of-order completion

Solution

RAW:

1) I2 needs the value of R3 which is written is I1. So, I2 is dependent on I1.
2) I4 needs the value of R3 which is written is I3. So, I4 is dependent on I3.
3) I7 needs the value of R4 which is written is I6. So, I7 is dependent on I6.

WAW:

1) WAW dependency will occur if I3 is completed before the completion of I1
2) WAW dependency will occur if I6 is completed before the completion of I2
3) WAW dependency will occur if I7 is completed before the completion of I4

WAR:

1) WAR dependency will occur if I3 is completed before the issue of I2
2) WAR dependency will occur if I3 is completed before the issue of I1
3) WAR dependency will occur if I4 is completed before the issue of I3
4) WAR dependency will occur if I5 is completed before the issue of I1
5) WAR dependency will occur if I6 is completed before the issue of I5
6) WAR dependency will occur if I6 is completed before the issue of I5
7) WAR dependency will occur if I7 is completed before the issue of I3

Removing constraints using register renaming:

I1: R3b ← R3a op R5a
I2: R4a ← R3b + 1
I3: R3c ← R7a + 1
I4: R7b ← R3c op R4a
I5: R5b ← R4a + 3
I6: R4b ← R3c op R6a
I7: R7c ← R4b + 4

Constraints:

1) I2 is dependent on I1.
2) I4 is dependent on I3.
3) I7 is dependent on I6.
4) I1 takes 2 clock cycles in the execution block
5) I4 takes 2 clock cycles in the execution block
6) I6 takes 2 clock cycles in the execution block

Out of order execution with out of order completion

Decode		Window	Execute		Write		Clock cycle
I1	I2						1
I3	I4	I1, I2	I1				2
I5	I6	I2, I3, I4	I1	I3			3
I7		I5, I6	I2	I4	I1	I3	4
		I6, I7	I5	I4	I2		5
		I7	I6		I4	I5	6
		I7	I6				7
			I7		I6		8
					I7		9