

Superscalar (II)

Register Renaming

- Register renaming is a method to address storage conflicts. Duplication of resources can address it.
- Output dependency and anti-dependency are examples of storage conflicts. So, they can be address using register renaming.
- In essence, registers are allocated dynamically by the processor hardware.
 - When a new register value is created, a new register is allocated for that value.
 - Subsequent instructions that access that value must be revised to refer to the new register containing the value. (Renaming)
 - The same original register reference in several different instructions may refer to different actual registers, if different values are intended.
- Let's use the same example used in the last class again:
 - I0: R3 op R5 -> R3
 - I1: R3 + 1 -> R4
 - I2: R5 + 1 -> R3
 - I3: R3 op R4 -> R7
- After applying register renaming, the revised instructions look like:
 - I0: R3 op R5 -> R3(a)
 - I1: R3(a) + 1 -> R4
 - I2: R5 + 1 -> R3(b)
 - I3: R3(b) op R4 -> R7
- The register reference with a pair parentheses refers to a hardware register allocated to hold a new value.
- Note: R3(a) and R3(b) avoid the WAR and WAW.

Exercise

- Given the instructions below, list all the dependencies (RAW, WAW, WAR) it has, and apply register renaming to address output dependency (WAW) and anti-dependency (WAR).
 - I0: R3 + 1 -> R3
 - I1: R3 + R2 -> R4
 - I2: R3 op R4 -> R7
 - I3: Store R0 -> R4

- True data dependency (RAW):
 - **I1 depends** on the result of **I0 (R3)**
 - **I2 depends** on the result of **I0 and I1 (R3, R4)**
- WAW: **I3** writes after **I1** write to R4
- WAR: **I3** writes after **I2** reads R4
- Register renaming:
 - I0: $R3 + 1 \rightarrow R3(a)$
 - I1: $R3(a) + R2 \rightarrow R4$
 - I2: $R3(a) \text{ op } R4 \rightarrow R7$
 - I3: Store R0 $\rightarrow R4(a)$

Instruction Issue Policy

- In essence, the processor is trying to **look ahead of current point** of execution to **locate instructions** that can be brought **into the pipeline**.
- Three types of ordering are important in this regards:
 - Order in which instructions are **fetched**
 - Order in which instructions are **executed** (constrained by data dependencies)
 - Order in which instructions **update** registers and memory values (order of completion)
- One **constraint: results must be correct**. So, the processor must **accommodate** the various **dependencies and conflicts** discussed earlier.
- Four categories:
 - In-order issue (order to execute), in-order completion (order to write the result)
 - In-order issue, out-of-order completion
 - Out-of-order issue, out-of-order completion
 - Out-of-order issue, in-order completion
- Example:

- Assume a superscalar pipeline is capable of fetching and decoding 2 instructions at a time
 - Instructions are fetched and decoded in pair. The next two instructions must wait to be decoded until the pair of decode pipeline stages has cleared.
- having 3 separate ALUs (e.g., two for integer arithmetic and one for floating-point arithmetic)
- 2 instances of the write-back pipeline stage
- 6 instruction code fragment with the following constraints:
 - I1 requires two cycles to execute
 - I3 and I4 conflict for the same ALU (e.g., both need floating-point arithmetic)
 - I5 depends on the value produced by I4
 - I5 and I6 conflict for an ALU (which may be different from the one I3 and I4 need)
- To fetch, decode, and write back an instruction, each stage need 1 clock cycle.
- When there is a conflict for a functional unit, or when a functional unit requires more than one cycle to generate a result, instructions temporarily stall.

In-order issue, in-order completion

- **Sequential execution** (in-order issue) and to write results in that same order (in-order completion)
- It is the simplest policy. **Not very efficient.** Instruction must **stall if necessary**.

Fetch		Decode		ALU1	ALU2	ALU3	WriteBack1	WriteBack2	Cycle
I1	I2								1
I3	I4	I1	I2						2
I5	I6	I3	I4	I1	I2				3
(I5)	(I6)	(I3)	(I4)	I1	(I2)	I3			4
(I5)	(I6)		(I4)			(I3)	I1	I2	5
		I5	I6			I4	I3		6
						I5	I4		7
						I6	I5		8
							I6		9

- Note: in this example, I5 can also use ALU1 or ALU2 at cycle 7, and I6 must use the same ALU as I5, as they compete for the ALU. I5 depends on the value produced by I4, so I5 has to wait until I4 finish execution in cycle 6.