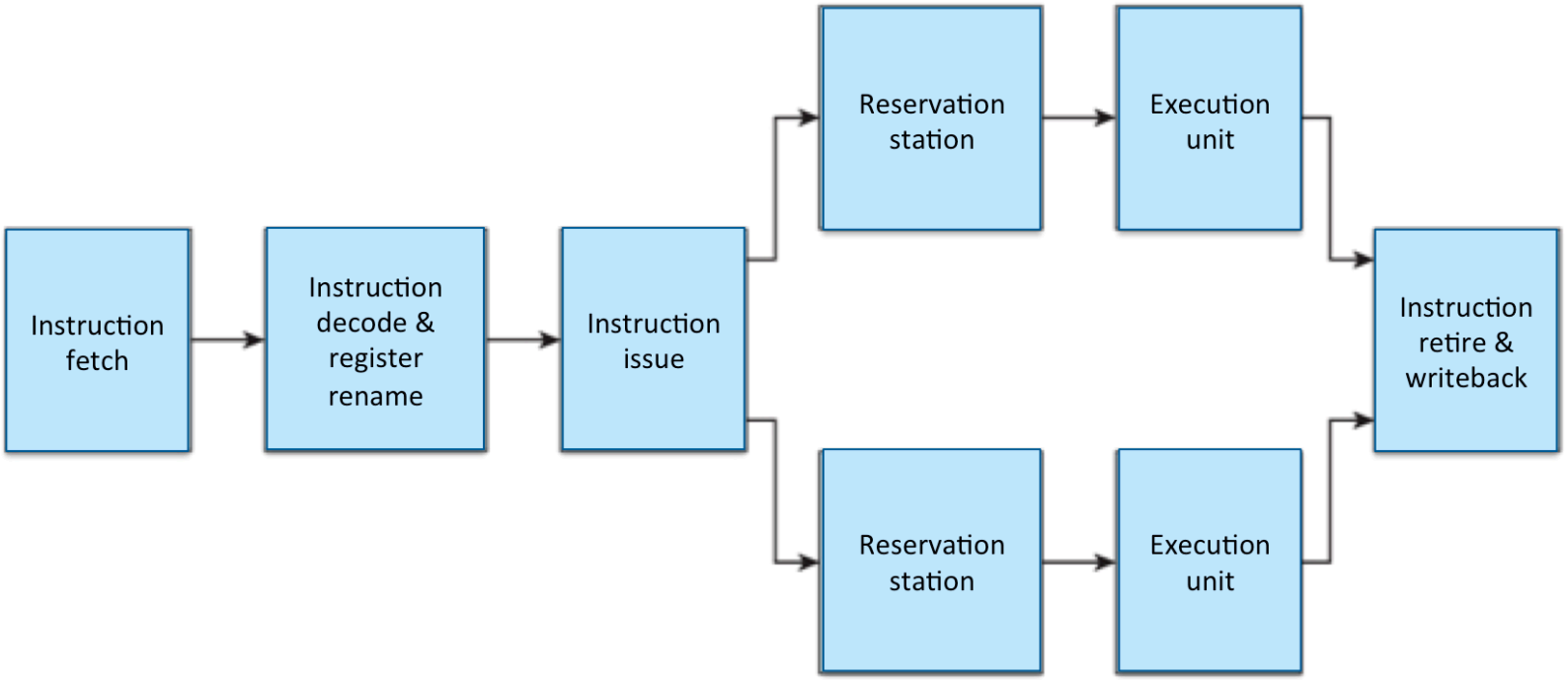
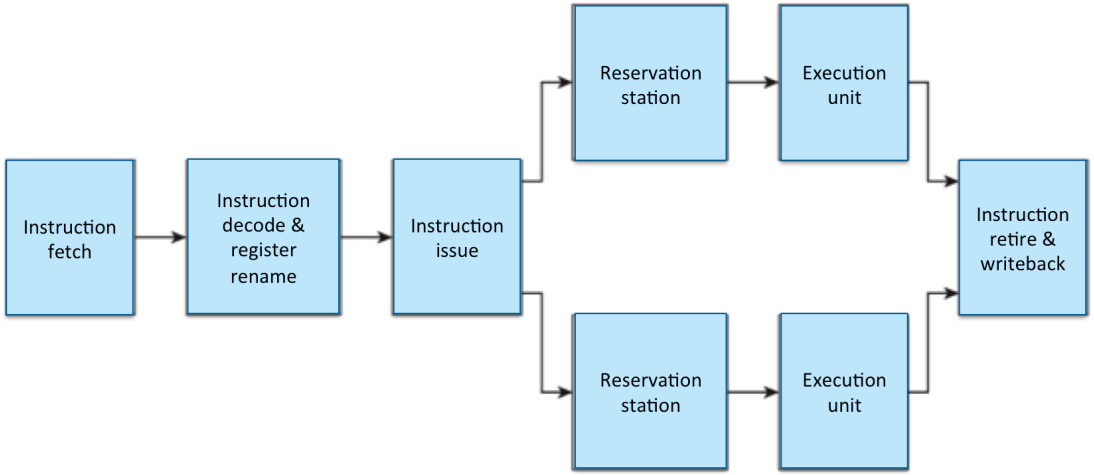
Superscalar Operation

A more realistic view of a superscalar system is:



Superscalar Operation



Instruction fetch

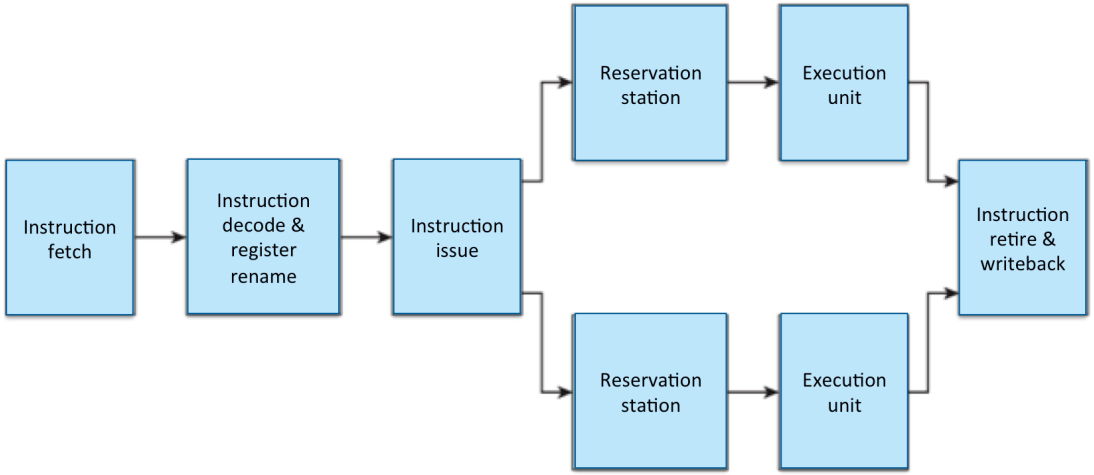
obtains instructions from cache or memory

Instruction Decode

interprets opcodes

substitutes temporary registers to avoid unnecessary stalls

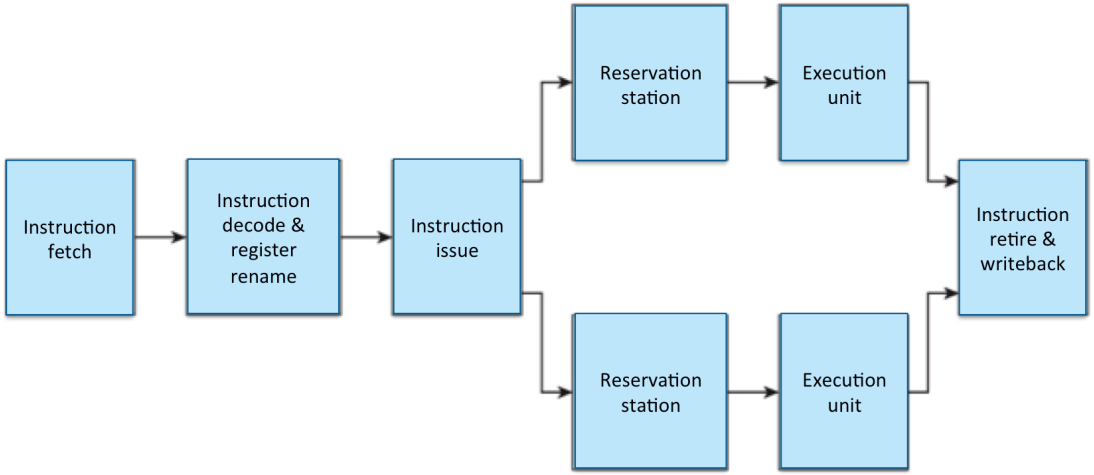
Superscalar Operation



Instruction issue

- ensures as many instructions as possible execute in parallel
- sends instructions to reservation stations (*issues* them)
- Instructions are *dispatched* from the reservation stations
  - once the required input operands are available

Superscalar Operation



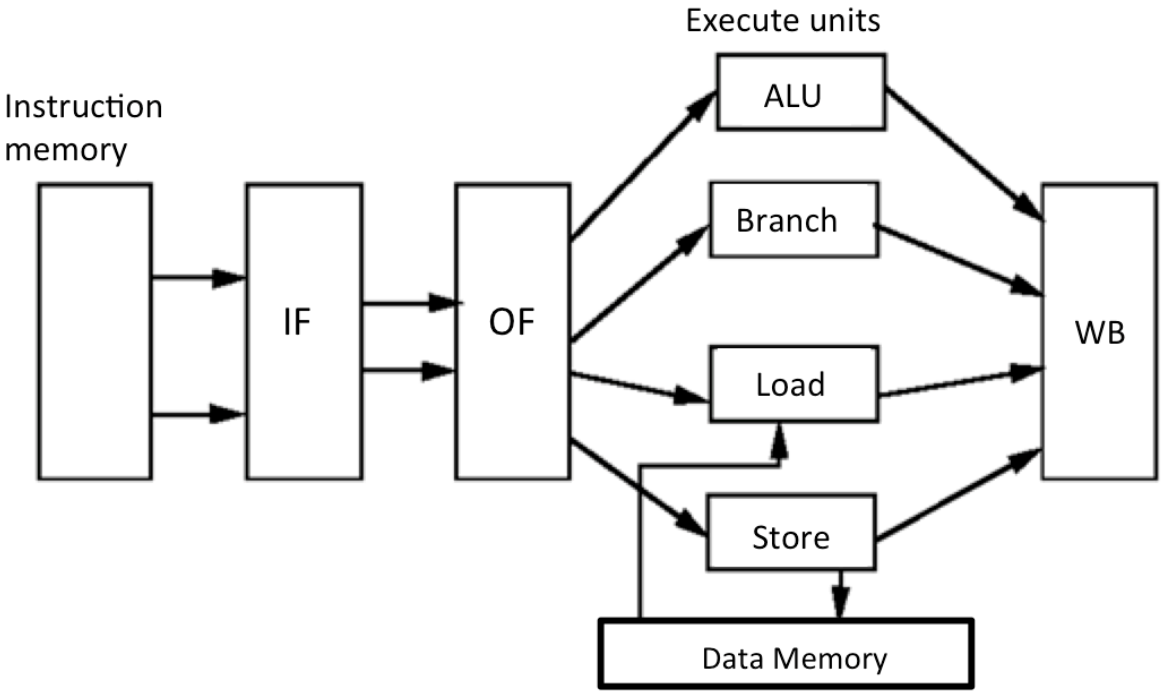
Reservation stations

- Serve as front end buffer to execution units
- Hold instructions that use the corresponding execution unit

Instruction retire

- Writes the results to the destination registers (*commits* them)
- Tells reservation stations when resources are available

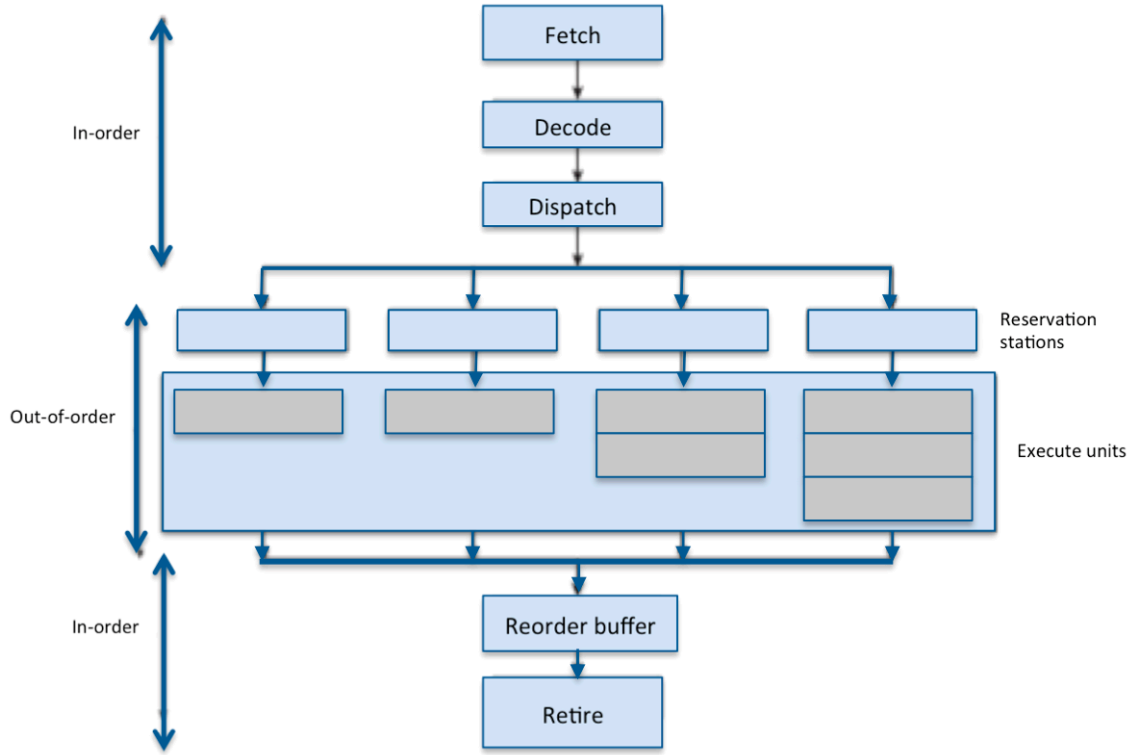
Superscalar Operation



Even with in-order issue, instructions may complete out-of-order

e.g. An add or shift may be issued after a lw or floating point instruction, but may finish executing first.

Superscalar Operation



Reorder buffers hold completed instructions

ensure that the program meaning or outcome is preserved

instructions that complete out-of-order are retired in-order

retire means writing the result to the correct register

Superscalar Operation

Possible issue/completion options:

- in-order issue & in-order completion
  - used in scalar pipeline
- in-order issue & out-of-order completion
- out-of-order issue & in-order completion
- out-of-order issue & out-of-order completion

Changing execution order can cause other types of data dependencies

Types of Dependencies

1. True data dependency (RAW) read after write

```
Sub  $3, $2, $4
Add  $5,$3,$4        # must read $5 after it is written by sub
```

2. Antidependency (WAR) write after read

```
or    $2, $4, $3
sub  $4, $5,$6     # can't write $4 until after the or reads $4
```

The or may be stalled waiting on a resource or on $3

Types of Dependencies

3. Output dependency (WAW) write after write

```
sub   $3, $2,$4
add   $5,$6,$5
slt   $3, $7,$0    # must write $3 after sub writes $3
```

Register Renaming

The use of registers can create apparent dependencies

1. or $3, $3,$5
2. add $4,$3,$2
3. add $3, $5, $2
4. sub $7,$3,$4

WAW exists between 3. and 1. (both write $3)

WAR exists between 3. and 2. (3. overwrites $3 input to 2.)

RAW exists between (1. & 2.), (2. & 4.), (3. & 4.)

Register Renaming

The use of registers can create apparent dependencies

1. or $3_a, $3_a,$5
2. add $4,$3_a,$2
3. add $3_b, $5, $2
4. sub $7,$3_b,$4

3_a and 3_b are unrelated, so a different register can be used for 3_b

- Eliminates false dependencies
- Register renaming performs this dynamic substitution

Register renaming adapts to larger register sets

- Applications use more registers without having to be recompiled
- Recompilation is required without register renaming feature