OOO Execution and 21264
Parallelism

• $ET = IC \times CPI \times CT$
• IC is more or less fixed
• We have shrunk cycle time as far as we can
• We have achieved a CPI of 1.
• Can we get faster?
Parallelism

- $ET = IC \times CPI \times CT$
- $IC$ is more or less fixed
- We have shrunk cycle time as far as we can
- We have achieved a CPI of 1.
- Can we get faster?

We can reduce our CPI to less than 1.
The processor must do multiple operations at once.
This is called Instruction Level Parallelism (ILP)
The Basic 5-stage Pipeline

- Like an assembly line -- instructions move through in lock step
- In the best case, it can achieve one instruction per cycle (IPC).
- In practice, it’s much worse -- branches, data hazards, long-latency memory operations cause much lower IPC.
- We want an IPC > 1!!!
Approach 1: Widen the pipeline

- Process two instructions at once instead of 1
- Often 1 “odd” PC instruction and 1 “even” PC
  - This keeps the instruction fetch logic simpler.
- 2-wide, in-order, superscalar processor
- Potential problems?
Dual issue: Structural Hazards

- Structural hazards
  - We might not replicate everything
  - Perhaps only one multiplier, one shifter, and one load/store unit
  - What if the instruction is in the wrong place?

If an “upper” instruction needs the “lower” pipeline, squash the “lower” instruction.
Dual issue: Structural Hazards

- Structural hazards
  - We might not replicate everything
  - Perhaps only one multiplier, one shifter, and one load/store unit
  - What if the instruction is in the wrong place?

If an “upper” instruction needs the “lower” pipeline, squash the “lower” instruction
Dual issue: Data Hazards

- The “lower” instruction may need a value produced by the “upper” instruction
- Forwarding cannot help us -- we must stall.
Compiling for Dual Issue

• The compiler should
  • Pair up non-conflicting instructions
  • Align branch targets (by potentially inserting noops above them)
• These are similar to the rules for VLIW, but they are just guidelines, not rules.
Beyond Dual Issue

- Wider pipelines are possible.
  - There is often a separate floating point pipeline.
- Wide issue leads to hardware complexity
- Compiling gets harder, too.
- In practice, processors use of two options if they want more ILP
  - If we can change the ISA: VLIW
  - If we can’t: Out-of-order
Going Out of Order: Data dependence refresher.

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or $t5,$t1,$t2
4: add $t3,$t1,$t2
Going Out of Order: Data dependence refresher.

There is parallelism!! We can execute 1 & 2 at once and 3 & 4 at once.

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or $t5,$t1,$t2
4: add $t3,$t1,$t2
Going Out of Order: Data dependence refresher.

There is parallelism!!
We can execute 1 & 2 at once and 3 & 4 at once

We can parallelize instructions that do not have a “read-after-write” dependence (RAW)

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or $t5,$t1,$t2
4: add $t3,$t1,$t2
Data dependences

• In general, if there is no dependence between two instructions, we can execute them in either order or simultaneously.

• But beware:
  • Is there a dependence here?
  
  1: add $t1,$s2,$s3

  2: sub $t1,$s3,$s4
  
  • Can we reorder the instructions?

  2: sub $t1,$s3,$s4

  1: add $t1,$s2,$s3

  • Is the result the same?
In general, if there is no dependence between two instructions, we can execute them in either order or simultaneously.

But beware:

- Is there a dependence here?

  1: add $t1,$s2,$s3

  2: sub $t1,$s3,$s4

- Can we reorder the instructions?

  2: sub $t1,$s3,$s4

  1: add $t1,$s2,$s3

- Is the result the same?

No! The final value of $t1$ is different
False Dependence #1

- Also called “Write-after-Write” dependences (WAW) occur when two instructions write to the same value.
- The dependence is “false” because no data flows between the instructions -- They just produce an output with the same name.
Beware again!

- Is there a dependence here?
  1: add $t1,$s2,$s3
  2: sub $s2,$s3,$s4

- Can we reorder the instructions?
  2: sub $s2,$s3,$s4
  1: add $t1,$s2,$s3

- Is the result the same?
Beware again!

• Is there a dependence here?
  1: add $t1, $s2, $s3
  2: sub $s2, $s3, $s4

• Can we reorder the instructions?
  2: sub $s2, $s3, $s4
  1: add $t1, $s2, $s3

• Is the result the same?

No! The value in $s2 that I needs will be destroyed
False Dependence #2

• This is a Write-after-Read (WAR) dependence
• Again, it is “false” because no data flows between the instructions
Out-of-Order Execution

- Any sequence of instructions has set of RAW, WAW, and WAR hazards that constrain its execution.
- Can we design a processor that extracts as much parallelism as possible, while still respecting these dependences?
The Central OOO Idea

1. Fetch a bunch of instructions
2. Build the dependence graph
3. Find all instructions with no unmet dependences
4. Execute them.
5. Repeat
Example

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or  $t3,$t1,$t2
4: add $t5,$t1,$t2
Example

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or  $t3,$t1,$t2
4: add $t5,$t1,$t2
Example

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or  $t3,$t1,$t2
4: add $t5,$t1,$t2
Example

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or $t3,$t1,$t2
4: add $t5,$t1,$t2
5: or $t4,$s1,$s3
6: mul $t2,$t3,$s5
7: sl $t3,$t4,$t2
8: add $t3,$t5,$t1
Example

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or $t3,$t1,$t2
4: add $t5,$t1,$t2
5: or $t4,$s1,$s3
6: mul $t2,$t3,$s5
7: sl $t3,$t4,$t2
8: add $t3,$t5,$t1
Example

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or $t3,$t1,$t2
4: add $t5,$t1,$t2
5: or $t4,$s1,$s3
6: mul $t2,$t3,$s5
7: sl $t3,$t4,$t2
8: add $t3,$t5,$t1
1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or  $t3,$t1,$t2
4: add $t5,$t1,$t2
5: or $t4,$s1,$s3
6: mul $t2,$t3,$s5
7: s1  $t3,$t4,$t2
8: add $t3,$t5,$t1
Example

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or $t3,$t1,$t2
4: add $t5,$t1,$t2
5: or $t4,$s1,$s3
6: mul $t2,$t3,$s5
7: sl $t3,$t4,$t2
8: add $t3,$t5,$t1

8 Instructions in 5 cycles
Simplified OOO Pipeline

- A new “schedule” stage manages the “Instruction Window”
- The window holds the set of instruction the processor examines
  - The fetch and decode fill the window
  - Execute stage drains it
- Typically, OOO pipelines are also “wide” but it is not necessary.

Impacts
- More forwarding, More stalls, longer branch resolution
- Fundamentally more work per instruction.
The Instruction Window

• The “Instruction Window” is the set of instruction the processor examines
  • The fetch and decode fill the window
  • Execute stage drains it
• The larger the window, the more parallelism the processor can find, but...
• Keeping the window filled is a challenge