Overview of Instruction-Level Parallelism

There are at least two different kinds of parallelism in computing.

- Using multiple processors to work toward a given goal, with each processor running its own program.
- Using only a single processor to run a single program, but allowing instructions from that program to execute in parallel.

The latter is called instruction-level parallelism, or ILP.

Several techniques are used in realizing ILP.

**Dynamic scheduling**

So far, we have seen that data hazards that prevent instruction issue were hidden by—

- Forwarding (bypasses)
- Compiler scheduling that separated dependent instructions.

The latter is referred to as static scheduling.

Dynamic scheduling is also possible:

The CPU rearranges the instructions (while preserving dependences) to reduce stalls.

Dynamic scheduling has several advantages over static:

- Handles dependences that are unknown at compile time (e.g., a memory reference to an address calculated at run time).
- Can adapt to dynamic branching behavior. (If a lot of branches are executed in a short period of time, compiler will not be able to schedule instructions efficiently for all cases.)
• Allows code compiled with one pipeline in mind to run efficiently on a different pipeline.

The idea of dynamic scheduling is to expose more parallelism by ignoring artificial serial constraints of “sequential” program, e.g., the one at the right.

This is also called out-of-order (OOO) execution: instructions don’t have to execute in original program order.

Early machines that used dynamic scheduling: CDC 6600 and IBM 360/91.

```
LW  R1, A
LW  R2, B
ADD R3, R1, R2
SW  R3, C
LW  R4, 8(R1)
LW  R5, 8(R2)
ADD R6, R4, R5
SW  R6, 8(R3)
LW  R7, 16(R1)
LW  R8, 16(R2)
ADD R9, R7, R8
SW  R9, 16(R3)
LW  R10, 24(R1)
LW  R11, 24(R2)
```

**Register renaming**

Suppose we have antidependences and output dependences:

```
ADD  R3, R2, R1
SUB  R1, R4, R5
MUL  R1, R6, R7
```

In the last lecture, we said that the compiler could simply use different registers for these three occurrences of R1.

This is called *register renaming*.

It can also be done by hardware—the hardware has a large set of behind-the-scenes unprogrammable registers, enough to hold several values of R1 at the same time.

Register renaming is often used with dynamic scheduling.

**Precise interrupts**

When an interrupt occurs, what happens to instructions in the pipeline?
What are the steps in handling an interrupt?

What problems does this pose for a pipelined system?

If instructions execute out of order, how can we keep from committing later instructions before earlier instructions have finished?

*Solution:* We use a reorder buffer, which provides a place to hold an instruction until we know that it can write its results.

The reorder buffer (ROB) is a set of registers that hold the result of an instruction between the time the operation finishes and the time the instruction commits.

This is similar to register renaming, where we hold values “behind the scenes” until it is time to commit them to a register.

**Superscalar and VLIW processors**

We can use caches, branch prediction, and branch target buffers to decrease the CPI of a machine to close to 1. If we want to reduce it further, we need to execute more than one instruction at a time.

But, we can’t get a CPI of less than one by *issuing* (beginning) only one instruction per cycle. So, we need to resort to *multiple issue*.

Multiple-issue processors come in two varieties.

- *Superscalar processors*, which issue varying numbers of instructions per clock cycle. They may be statically or dynamically scheduled.
- **VLIW (Very Long Instruction Word) processors** use instructions that, essentially, contain multiple opcodes.

In the Multiflow Trace series, each processor has an integer (I) unit and a floating-point (F) unit. A machine can have 1, 2, or 4 I-F pairs.

As shown below, each I-F pair is controlled by a 256-bit instruction.

**Word 0: I 0, ALU 0, early beat.**

```
<table>
<thead>
<tr>
<th>31</th>
<th>25 24</th>
<th>19 18</th>
<th>16 15</th>
<th>13 12 11</th>
<th>7 6</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>dest</td>
<td>dest</td>
<td>branch</td>
<td>test</td>
<td>src1</td>
<td>src2</td>
</tr>
</tbody>
</table>
```

**Word 1: immediate constant 0 (early).**

```
<table>
<thead>
<tr>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

**Word 2: I 0, ALU 1, early beat.**

```
<table>
<thead>
<tr>
<th>31</th>
<th>25 24</th>
<th>19 18</th>
<th>16 15</th>
<th>13 12 11</th>
<th>7 6</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>dest</td>
<td>dest</td>
<td>branch</td>
<td>test</td>
<td>src1</td>
<td>src2</td>
</tr>
</tbody>
</table>
```

**Word 3: F 0 FA/ALUA control fields.**

```
<table>
<thead>
<tr>
<th>31</th>
<th>25 24 23 22</th>
<th>17 16 15</th>
<th>11 10</th>
<th>6 5 4 3</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>64</td>
<td>dest</td>
<td>src1</td>
<td>src2</td>
<td>dest</td>
</tr>
</tbody>
</table>
```

**Word 4: I 0, ALU 0, late beat.**

```
<table>
<thead>
<tr>
<th>31</th>
<th>25 24</th>
<th>19 18</th>
<th>16 15</th>
<th>13 12 11</th>
<th>7 6</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>dest</td>
<td>dest</td>
<td>branch</td>
<td>test</td>
<td>src1</td>
<td>src2</td>
</tr>
</tbody>
</table>
```

**Word 5: immediate constant 0 (late).**

```
<table>
<thead>
<tr>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

**Word 6: I 0, ALU 1, late beat.**

```
<table>
<thead>
<tr>
<th>31</th>
<th>25 24</th>
<th>19 18</th>
<th>16 15</th>
<th>13 12 11</th>
<th>7 6</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>dest</td>
<td>dest</td>
<td>branch</td>
<td>test</td>
<td>src1</td>
<td>src2</td>
</tr>
</tbody>
</table>
```

**Word 7: F 0 FM/ALUM control fields.**

```
<table>
<thead>
<tr>
<th>31</th>
<th>25 24 23 22</th>
<th>17 16 15</th>
<th>11 10</th>
<th>6 5 4 3</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>64</td>
<td>dest</td>
<td>src1</td>
<td>src2</td>
<td>dest</td>
</tr>
</tbody>
</table>
```
On each cycle, a functional unit can perform one floating-point operation or two integer operations. One integer operation is performed on the first subcycle (“early beat”) and one on the second subcycle (“late beat”).

**True data dependences**

Consider our simple 5-stage pipeline

- Most operations take only 1 cycle to execute
- Bypasses handle most RAW hazards optimally (i.e., no stalls).
- Loads are a problem.
  - Cache hit: 2 cycles (EX generates address, MEM loads data)
  - Cache miss: Takes forever
  - A data-dependent instruction must stall.
  - Worse: The entire pipeline stalls, even other independent instructions.

Longer latencies make matters worse, e.g., floating-point operations.

Superscalar processing makes matters worse. Even with short latencies, *long serial dependence chains* limit parallelism.

**Dynamic scheduling**

*Goal:* Lessen impact of true data dependences

- Only data dependent instructions should have to wait for data (stall).
- Independent instructions after the dependent ones don’t stall if their data is available.
- Execute instructions in the order specified by true data dependences, not the “artificial” program order.
• Dataflow graph defines execution order

Program fragment

```
DDIV  F0, F2, F4
DADD  F10, F0, F8
DSUB  F8, F8, F14
```

Dataflow Graph

![Dataflow Graph](image)

Division takes many cycles: stall the dependent DADD but not the independent DSUB.

Out-of-order (OOO) execution: DSUB executes before DADD.

**Pipeline modification**

To attain OOO execution, a fundamental pipeline change is needed:

- Stalled dependent instructions must not be allowed to block later independent instructions

To achieve this, split ID (decode stage) into two new stages:

- **DISPATCH (ID):** instruction dispatch.
- **ISSUE (IS):** instruction schedule/issue.

The ISSUE stage buffers instructions waiting (stalled) for data

This allows later instructions to make progress (IF and DISPATCH are not stalled).

We start with a simple in-order pipeline:

```
IF → ID → EX → MEM → WB
```

In this pipeline, a stalled instruction has nowhere to go.

It stalls in the decode (ID) stage. This stalls a later instruction in IF.
Out-of-order pipeline

Two decoupled pipelines:

- **FETCH/DISPATCH** pipeline brings instructions into processor in program order.
  
  Scan instructions in program order—required to determine data dependences.

- **ISSUE/EXECUTE** pipeline executes instructions based on data dependences and available functional units (FU).

Here is a diagram showing the functional units on the CDC 6600: