# Chapter 6

# Instruction Level Parallelism

Most of the modern processors are designed to execute multiple instructions at the same time in order to improve the execution speed. In this chapter, we study a number of techniques to achieve the *Instruction Level Parallelism.*

## 6.1 Superpipelining

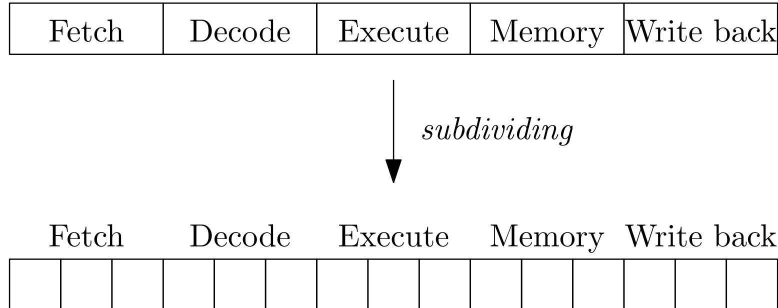
Similar to the multi-cycle implementation, the clock period in the pipelined implementation is set based on the *slowest stage* in the pipeline, i.e.

$$P = \max\{P_f, P_d, P_e, P_m, P_{wb}\}$$

Different from the multi-cycle implementation, the processor completes one instruction every cycle. Or, we can say that CPI of the pipelined implementation is approximately 1.

Therefore, *shortening* the clock period is a way to improve the performance. This can be done by *subdividing* each pipeline stage.

**Example 6.1** Suppose we can subdivide each stage of the pipeline into 3 substages.

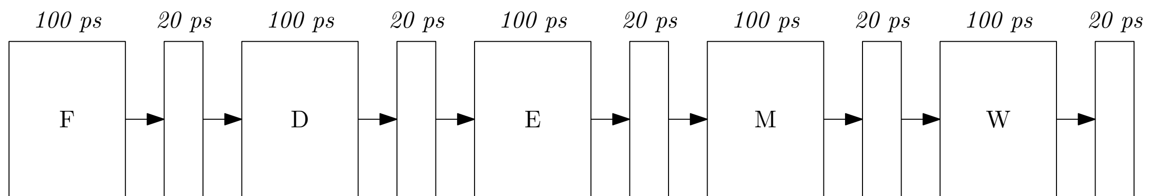



| Instr. \ Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | F | D | E | M | W | | | | | |
| 2 | | F | D | E | M | W | | | | |
| 3 | | | F | D | E | M | W | | | |
| 4 | | | | F | D | E | M | W | | |
| 5 | | | | | F | D | E | M | W | |
| 6 | | | | | | F | D | E | M | W |

| Instr. \ Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | $F_1$ | $F_2$ | $F_3$ | $D_1$ | $D_2$ | $D_3$ | $E_1$ | $E_2$ | $E_3$ | $M_1$ | $M_2$ | $M_3$ | $W_1$ | $W_2$ | $W_3$ | | | | | |
| 2 | | $F_1$ | $F_2$ | $F_3$ | $D_1$ | $D_2$ | $D_3$ | $E_1$ | $E_2$ | $E_3$ | $M_1$ | $M_2$ | $M_3$ | $W_1$ | $W_2$ | $W_3$ | | | | |
| 3 | | | $F_1$ | $F_2$ | $F_3$ | $D_1$ | $D_2$ | $D_3$ | $E_1$ | $E_2$ | $E_3$ | $M_1$ | $M_2$ | $M_3$ | $W_1$ | $W_2$ | $W_3$ | | | |
| 4 | | | | $F_1$ | $F_2$ | $F_3$ | $D_1$ | $D_2$ | $D_3$ | $E_1$ | $E_2$ | $E_3$ | $M_1$ | $M_2$ | $M_3$ | $W_1$ | $W_2$ | $W_3$ | | |
| 5 | | | | | $F_1$ | $F_2$ | $F_3$ | $D_1$ | $D_2$ | $D_3$ | $E_1$ | $E_2$ | $E_3$ | $M_1$ | $M_2$ | $M_3$ | $W_1$ | $W_2$ | $W_3$ | |
| 6 | | | | | | $F_1$ | $F_2$ | $F_3$ | $D_1$ | $D_2$ | $D_3$ | $E_1$ | $E_2$ | $E_3$ | $M_1$ | $M_2$ | $M_3$ | $W_1$ | $W_2$ | $W_3$ |

**Exercise 6.1** What is the throughput, the number of instructions executed per second, of the following pipelined implementation?

Here, each stage requires 100 picoseconds for its task, and a temporary storage requires additional 20 picoseconds to save the result.




What are the throughputs when we subdivide each stage into 3 and 5 substages, respectively?

Table 6.1: Pipeline Depths in Real-world Processors

| Processors | Depth |
|---|---|
| UltraSPARC T1 | 6 |
| PowerPC G4e | 7 |
| UltraSPARC T2, SPARC T3, ARM Cortex A9 | 8 |
| AMD Athlon, Qualcomm Scorpion | 10 |
| Qualcomm Krait | 11 |
| Intel Pentium Pro/II/III, AMD Athlon 64/Phenom | 12 |
| Apple A6 | 12 |
| Nvidia Denver | 13 |
| UltraSPARC III/IV, Intel Core 2, Apple A7/A8 | 14 |
| Intel Core Gen 2, 3, 4, 5, 6, 7 | 14/19 |
| ARM Core A15/A57 | 15 |
| PowerPC G5, Intel Core Gen 1 | 16 |
| AMD Bulldozer/Piledriver/StreamRoller | 18 |
| Pentium 4 | 20 |
| Pentium 4E | 31 |

## 6.2 Multiple Issue

Another approach to improve the performance is to replicate the pipeline stages. Therefore, multiple instructions can be executed at the same time. This technique is called *multiple issue*.

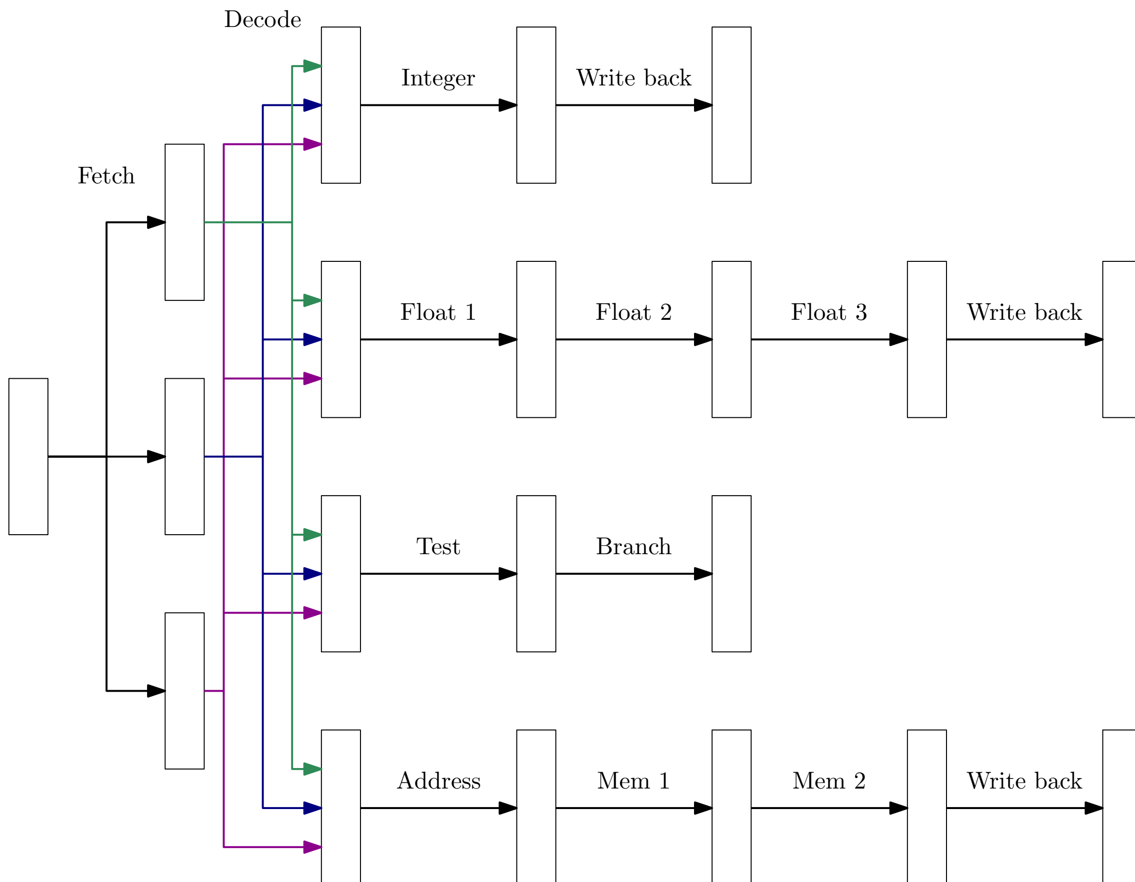
| Instr. \ Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | F | D | E | M | W | | | | | |
| 2 | | F | D | E | M | W | | | | |
| 3 | | | F | D | E | M | W | | | |
| 4 | | | | F | D | E | M | W | | |
| 5 | | | | | F | D | E | M | W | |
| 6 | | | | | | F | D | E | M | W |

| Instr. \ Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | F | D | E | M | W | | | | | |
| 2 | F | D | E | M | W | | | | | |
| 3 | | F | D | E | M | W | | | | |
| 4 | | F | D | E | M | W | | | | |
| 5 | | | F | D | E | M | W | | | |
| 6 | | | F | D | E | M | W | | | |

However, replicating the stages does not make the throughput of the process drastically increase. For example, having two copies of the pipeline stages does not double the instruction execution speed. This is because of the dependency among instructions. Here, any type of dependencies, i.e. RAW, WAR, and WAW, may cause data hazards in the execution.

Moreover, it may be more cost effective when parts of the pipeline stages are replicated.

The following processor can issue 3 instructions in one cycle. However, they have four different function units: (1) integer, (2) floating-point computation, (3) branch, and (4) memory.

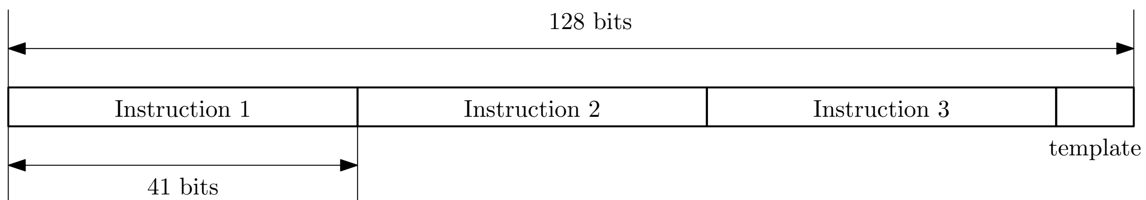


### 6.2.1 Very Long Instruction Word (VLIW)

VLIW is an approach to handle data hazards in multiple-issue processors. It is done by grouping a number of instructions that can be performed at the same time into one *issue packet*. This grouping is based on the structure of the pipeline.

The processor then execute all the instructions in the packet without checking the dependency between the instructions. To combine instructions into a packet, we need to redesigned the instruction format to make a very long instruction. This becomes the name of this approach.

Intel Itanium is the first realization of VLIW. It was initially designed to be a successor of the IA-32 architecture. However, it is not popular since it is not backward compatible. Itanium is designed to fetch bundles of instructions. Each bundle is 128 bits long, and composes of 3 instructions.




A special compiler is needed to check the dependencies and reorder the instructions to maximize the performance of the processor. Having the compiler reorder the instructions makes the processor become less complicated since it does not a functional unit to check the dependencies between instructions. Moreover, it makes the processor consume less power. However, the machine code will depend on the microarchitecture of the processor. It is not easy to execute the same machine code on different processors.

**Exercise 6.2** A processor has two functional units: (1) ALU or branch instruction, (2) memory instruction. It can fetch a bundle of two instructions at a time. Reorder the instructions to avoid as many pipeline stalls as possible. Assume branches are predicted, so that control hazards are handled by the hardware.

`mrmov` have a *use latency of one clock cycle*, which prevents one instruction from using the result without stalling.

```
0x00 mrmov (%rcx), %r8
0x0A add   %rdx, %r8
0x0C rmmov %r8, (%rcx)
0x16 add   $-8, %rcx
0x18 bne   0x00
0x21 hlt
```

| Cycle | ALU/Branch | Memory |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

## Loop Unrolling

From the previous exercise, we can improve the speed of the program by making a four copies of the loop body. This reduces the *branch penalties*, and the unnecessary overhead. This technique is called *loop unrolling*.

Normal loop

```
int i;
for(i=0; i<100; i++) {
  A[i] += 10;
}
```

After loop unrolling

```
int i;
for(i=0; i<100; i+=4) {
  A[i] += 10;
  A[i+1] += 10;
  A[i+2] += 10;
  A[i+3] += 10;
}
```

**Example 6.2** Show how the loop unrolling help improve the execution speed. Assume that the number of elements is a multiple of four.

```
0x00 mrmov (%rcx), %r8
0x0A add   %rdx, %r8
0x0C rmmov %r8, (%rcx)
0x16 add   $-8, %rcx
0x18 bne   0x00
0x21 hlt
```

| Cycle | ALU/Branch | Memory |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

### Register Renaming

In the previous example, additional registers are required to perform loop unrolling. Those registers are automatically assigned by the compiler. This technique is called *register renaming*.

The *register renaming* technique can be used to eliminated the WAW and WAR dependencies. For example,

```
1) add   %rcx, %rdx
2) irmov $10, %rdx   ----> irmov $10, %r8
3) sub   %rax, %rdx  ----> sub   %rax, %r8
```

```
1) add   %rcx, %rdx
2) irmov $10, %rcx   ----> irmov $10, %r8
```

**Exercise 6.3** Renaming registers in the following program to eliminate WAW and WAR dependencies.

| | Before renaming | After renaming |
|---|---|---|
| i1) | irmov $10, %rax | |
| i2) | rrmov %rax, %rbx | |
| i3) | mrmov 4(%rax), %rcx | |
| i4) | add %rcx, %rdx | |
| i5) | push %rdx | |
| i6) | pop %rax | |

### 6.2.2 Superscalar

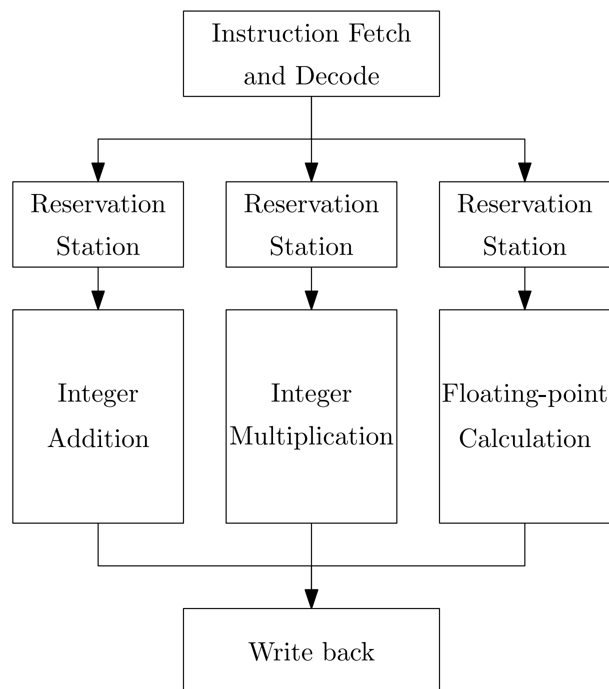
Superscalar is used to refer to a multiple-issue processor equipped with a hardware component for *dynamic pipeline scheduling*. It has an advantage over the VLIW processors that the same program can be shared among different microarchitectures with recompilation. However, the additional hardware component consumes more power to execute programs.

There are three different ways to execute programs with the dynamic pipeline scheduling.

1. In-order issue with in-order completion
2. In-order issue with out-of-order completion
3. Out-of-order issue with out-of-order completion

**Example 6.3** Given a superscalar processor with the following settings:

- The processor can fetch and decode 2 instructions at a time,
- It is equipped with 3 functional units i.e. integer adder, integer multiplier, and floating-point computation,
- Two instructions can be completed at a time,
- The floating-point calculation requires 2 cycles for the execution.




Show how to dynamically execute the following instructions using three different policies.

```
i1) fadd f2, f1    // f1 = f1 + f2
i2) add  r1, r2    // r2 = r2 + r1
i3) mul  r3, r4    // r4 = r4 * r3
i4) mul  r5, r6    // r6 = r6 * r5
i5) add  r6, r7    // r7 = r7 + r6
i6) add  r8, r9    // r9 = r9 + r8
```

*In-order issue with in-order completion*

Instructions are issued according to their orders in the program. An instruction cannot be issued before its previous ones has been issued, and it cannot be completed before its previous ones has been completed.

```
i1) fadd f2, f1    // f1 = f1 + f2
i2) add  r1, r2    // r2 = r2 + r1
i3) mul  r3, r4    // r4 = r4 * r3
i4) mul  r5, r6    // r6 = r6 * r5
i5) add  r6, r7    // r7 = r7 + r6
i6) add  r8, r9    // r9 = r9 + r8
```

| Instr. | Cycle | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| i1 | | | | | | | | |
| i2 | | | | | | | | |
| i3 | | | | | | | | |
| i4 | | | | | | | | |
| i5 | | | | | | | | |
| i6 | | | | | | | | |

*In-order issue with out-of-order completion*

Instructions are issued according to their orders in the program. But an instruction may complete before its previous ones when the instruction does not depend on the previous instructions. A *resource conflict* and a *data dependency* may cause the instruction issuing to be stalled.

```
i1) fadd f2, f1    // f1 = f1 + f2
i2) add  r1, r2    // r2 = r2 + r1
i3) mul  r3, r4    // r4 = r4 * r3
i4) mul  r5, r6    // r6 = r6 * r5
i5) add  r6, r7    // r7 = r7 + r6
i6) add  r8, r9    // r9 = r9 + r8
```

| Instr. | Cycle | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| i1 | | | | | | | | |
| i2 | | | | | | | | |
| i3 | | | | | | | | |
| i4 | | | | | | | | |
| i5 | | | | | | | | |
| i6 | | | | | | | | |

*Out-of-order Issue with out-of-order completion*

Out-of-order issue allows instructions to be decoded continuing regardless of the resource conflict and dependency. After the processor finishes decoding instructions, they are saved in a buffer called *instruction window*. The instructions are then issued into the execute stage when the particular functional units are available, and no dependencies block them.

```
i1) fadd f2, f1    // f1 = f1 + f2
i2) add  r1, r2    // r2 = r2 + r1
i3) mul  r3, r4    // r4 = r4 * r3
i4) mul  r5, r6    // r6 = r6 * r5
i5) add  r6, r7    // r7 = r7 + r6
i6) add  r8, r9    // r9 = r9 + r8
```

| Instr. | Cycle | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| i1 | | | | | | | | |
| i2 | | | | | | | | |
| i3 | | | | | | | | |
| i4 | | | | | | | | |
| i5 | | | | | | | | |
| i6 | | | | | | | | |

Table 6.2: Issue width in Real-world Processors

| Processors | Issue width |
|---|---|
| UltraSPARC T1 | 1 |
| UltraSPARC T2, SPARC T3, ARM Cortex A9 | 2 |
| Qualcomm Scorpion | 2 |
| Intel Pentium Pro/II/III/M, Pentium 4 | 3 |
| Qualcomm Krait, Apple A6 | 3 |
| ARM Core A15/A57 | 3 |
| UltraSPARC III/IV, PowerPC G4e | 4 |
| AMD Bulldozer/Piledriver/StreamRoller | 4 |
| PowerPC G5 | 5 |
| AMD Athlon 64/Phenom,Core 2, Core Gen 2,3 | 6 |
| Apple A7/A8 | 6 |
| Nvidia Denver | 7 |
| Intel Core Gen 4, 5, 6, 7 | 8 |

## 6.3 Branch Instructions

### 6.3.1 Predicated Instruction

Since conditional branch instructions may cause control hazards and stalls the execution for a number of cycles, a new type of instruction is introduced into the ISA, i.e. *conditional move*. This instruction is executed in the same manner as the *move* instruction. But the conditional move instruction is committed only when the condition is true.

Conditional Branch

```
    cmp %rcx, %rax
    jg  L2
    mov %r8, %r10
L2:
```

Conditional Move

```
    cmp    %rcx, %rax
    cmovle %r8, %r10
```

### 6.3.2 Speculative Execution

*Speculation execution* is another approach to handle the control hazards. In this approach, the processor immediately fetches and starts executing the instructions based on the outcome of the branch prediction. But the results will not be committed until the outcome of the branch is known.

However, if the prediction is incorrect, the processor needs to flush the pipeline, and reexecute the correct sequence of instructions.