CENG3420 Computer Organization & Design

Ryan Chan

May 3, 2025

Abstract

This is a note for **CENG3420 Computer Organization & Design**.
Contents are adapted from the lecture notes of CENG3420, prepared by Bei Yu, as well as some online resources.
This note is intended solely as a study aid. While I have done my best to ensure the accuracy of the content, I do not take responsibility for any errors or inaccuracies that may be present. Please use the material thoughtfully and at your own discretion.
If you believe any part of this content infringes on copyright, feel free to contact me, and I will address it promptly.
Mistakes might be found. So please feel free to point out any mistakes.

Contents

Chapter 1

Introduction

This course is about how computers work.

1.1 The Manufacturing Process of Integrated Circuit

For this chapter, only a few calculations need to be considered:

1. Yield $=$ The proportion of working dies per wafer.
2. Cost per die $= \dfrac{\text{Cost per wafer}}{\text{Dies per wafer} \ \times \ \text{Yield}}$
3. Dies per wafer $\approx \dfrac{\text{Wafer area}}{\text{Die area}}$ (since wafers are circle)
4. Yield $= \dfrac{1}{\left[1 + (\frac{\text{Defects per area} \times \text{Die area}}{2})\right]^2}$

> **Remark.** Note that the defects on average $=$ Defects per unit area $\times$ Die area.

1.2 Power

$$\text{Power} \ = \ \text{Capacitive load} \ \times \ \text{Voltage}^2 \times \ \text{Frequency}$$

> **Example.** For a simple processor, the capacitive load is reduced by 15%, voltage is reduced by 15%, and the frequency remains the same. Then, how much power consumption can be reduced?
>
> **Solution:**
>
> $$1 - (1 - 15\%) \times (1 - 15\%) \times 1 = 27.75\%$$
>
> Thus, 27.75% of the power consumption can be reduced.

Chapter 2

Instruction Set Architecture (ISA)

2.1 Organization

Computer components include the processor, input, output, memory, and network. The primary focus of this course is on the processor and its interaction with the memory system. However, it is impossible to understand their operation by examining each transistor individually due to their enormous quantity. Therefore, abstraction is necessary.

Both the control unit and datapath need circuitry to manipulate instructions — for example, deciding the next instruction, decoding, and executing instructions.

There is also system software, such as the operating system and compiler, which translate programs written in high-level languages into machine instructions.

For example, after a program is written in a high-level language (like C), the compiler translates it into assembly language. Then, the assembler converts the assembly code into machine code (object code). The machine code is stored in memory, and the processor's control unit fetches an instruction from memory, decodes it to determine the operation, and signals the datapath to execute the instruction. The processor then fetches the next instruction from memory, and this cycle repeats.

2.2 Instruction Set Architecture

The instruction set architecture (ISA) is the bridge between hardware and software. It is the interface that separates software from hardware and includes all the information necessary to write a machine language program, such as instructions, registers, memory access, I/O, etc.

To put it simple, ISA is a formal specification of the instruction set that is implemented in the machine hardware. It defines how software can control the hardware by specifying the instructions, registers, memory addressing modes, and I/O operations that the processor can execute.

Assembly language instructions are the language of the machine. We aim to design an ISA that makes it easy to build hardware and compilers while maximizing performance and minimizing cost. Therefore, in this course, we focus on the RISC-V ISA.

In a Reduced Instruction Set Computer (RISC), we have fixed instruction lengths, a load-store instruction set, and a limited number of addressing modes and operations. Thus, it is optimized for speed.

There are four design principles in RISC-V:

1. Simplicity favours regularity.
2. Smaller is faster.
3. Make the common case fast.
4. Good design demands good compromises.

2.3 RISC-V

There are five Instruction Categories:

1. Load and Store instruction
2. Bitwise instructions
3. Arithmetic instructions
4. Control transfer instructions
5. Pseudo instructions

	31 25	24 20	19 15	14 12	11 7	6 0
R-Type	funct7	rs2	rs1	funct3	rd	opcode

	31 20	19 15	14 12	11 7	6 0
I-Type	imm[11:0]	rs1	funct3	rd	opcode

	31 25	24 20	19 15	14 12	11 7	6 0
S-Type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

	31 12	11 7	6 0
U-Type	imm[31:12]	rd	opcode

Register names	ABI Names	Description
x0	zero	Hard-Wired Zero
x1	ra	Return Address
x2	sp	Stack Pointer
x3	gp	Global Pointer
x4	tp	Thread Pointer
x5	t0	Temporary / Alternate Link Register
x6-7	t1 - t2	Temporary Register
x8	s0 / fp	Saved Register / Frame Pointer
x9	s1	Saved Register
x10-11	a0 - a1	Function Argument / Return Value Registers
x12-17	a2 - a7	Function Argument Registers
x18-27	s2 - s11	Saved Register
x28-31	t3 - t6	Temporary Register

Chapter 3

Arithmetic Instructions

3.1 Introduction to RISC-V

Previously, we had the RV32I Unprivileged Integer Register table:

Register names	ABI Names	Description
x0	zero	Hard-Wired Zero
x1	ra	Return Address
x2	sp	Stack Pointer
x3	gp	Global Pointer
x4	tp	Thread Pointer
x5	t0	Temporary / Alternate Link Register
x6-7	t1 - t2	Temporary Register
x8	s0 / fp	Saved Register / Frame Pointer
x9	s1	Saved Register
x10-11	a0 - a1	Function Argument / Return Value Registers
x12-17	a2 - a7	Function Argument Registers
x18-27	s2 - s11	Saved Register
x28-31	t3 - t6	Temporary Register

There are some important registers to note:

Return address (ra): Used to save the function return address, usually PC + 4.

Stack pointer (sp): Holds the base address of the stack. It must be aligned to 4 bytes.

Global pointer (gp): Holds the base address of the location where global variables reside.

Argument registers (a0–a7): Used to pass arguments to functions.

Also, we have the RV32I base types:

	31 25	24 20	19 15	14 12	11 7	6 0
R-Type	funct7	rs2	rs1	funct3	rd	opcode

	31 20	19 15	14 12	11 7	6 0
I-Type	imm[11:0]	rs1	funct3	rd	opcode

	31 25	24 20	19 15	14 12	11 7	6 0
S-Type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

	31 12	11 7	6 0
U-Type	imm[31:12]	rd	opcode

Here, the opcode (7 bits) specifies the operation. rs1 (5 bits) is the register file address of the first source operand. rs2 (5 bits) is the register file address of the second source operand. rd (5 bits) is the register file address of the destination for the result. imm (12 bits or 20 bits) is the immediate value field. funct (3 bits or 10 bits) is the function code that augments the opcode.

Note that the rs1 and rs2 fields are kept in the same place, which causes the imm field in S-type instructions to be separated into two parts.

3.2 Arithmetic and Logical Instructions

Here, we introduce some simple arithmetic and logical instructions.

3.2.1 Arithmetic Instructions

In RISC-V, each arithmetic instruction performs a single operation and specifies exactly three operands, all of which are contained in the datapath's register file.

For example, we have:

Code 3.2.1.

```
add t0, a1, a2   # t0 = a1 + a2
sub t0, a1, a2   # t0 = a1 - a2
```

which can be understood as:

```
destination = source1 op source2
```

These instructions follow the R-type format.

3.2.2 Immediate Instructions

Small constants are often used directly in typical assembly code to avoid load instructions. RISC-V provides special instructions that contain constants. For example:

Code 3.2.2.

```
addi sp, sp, 4    # sp = sp + 4
slti t0, s2, 15   # t0 = 1 if s2 < 15
```

These instructions follow the I-type format. The constants are embedded within the instructions, limiting their values to the range from -2^{11} to $2^{11} - 1$.

Example.

```
.global _start

.text
_start:
        li a1, 20
        li a2, 23
        add t0, a1, a2
        sub t1, a1, a2
```

This will give the result:
`t0 = 0x2b, t1 = 0xfffffffd`

Note. The calculation of `t1` involves two's complement, which will be introduced later.

If we want to load a 32-bit constant into a register, we must use two instructions:

Code 3.2.3.

```
lui t0, 1010 1010 1010 1010 1010b
ori t0, t0, 1010 1010 1010b
```

Here, `lui` loads the upper 20 bits with an immediate value, and `ori` sets the lower 12 bits using an immediate value.

If a number is signed, then `1000 0000 ...` represents the most negative value, and `0111 1111 ...` represents the most positive value, since the first bit is used to distinguish between signed and unsigned values.

3.2.3 Shift Operations

We need operations to pack and unpack 8-bit characters into a 32-bit word, and we can achieve this by using shift operations. We can shift all the bits left or right:

Code 3.2.4.

```
slli t2, s0, 8   # t2 = s0 << 8 bits
srli t2, s0, 8   # t2 = s0 >> 8 bits
```

These instructions follow the I-type format. The above shifts are called logical because they fill the vacancy with zeros. Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 - 1$ or 31 bit positions.

Example.

```
1  .global _start
2
3  .text
4  _start:
5          li a1, 20
6          li a2, 23
7          slli t0, a1, 2
8          srli t1, a1, 1
```

```
Line 7: 10100 -> 1010000   # after slli 2 bits
Line 8: 10111 -> 01011     # after srli 1 bits
```

3.2.4 Logical Operations

There are numbers of bitwise logical operations in RISC-V ISA. For example:

R format:

Code 3.2.5.

```
and t0, t1, t2   # t0 = t1 & t2
or  t0, t1, t2   # t0 = t1 | t2
xor t0, t1, t2   # t0 = t1 & (not t2) + (not t1) & t2
```

I format:

Code 3.2.6.

```
andi t0, t1, 0xFF00   # t0 = t1 & 0xFF00
ori  t0, t1, 0xFF00   # t0 = t1 | 0xFF00
```

Example.

```
1   .global _start
2
3   .text
4   _start:
5           li a1, 20
6           li a2, 23
7           and t0, a1, a2
8           or t1, a1, a2
9           xor t2, a1, a2
10          andi t3, a1, 0x12
11          ori t4, a2, 0x21
```

```
a1 = 10100, a2 = 10111
Line 7:   t0 = 10100 & 10111  ->  10100
Line 8:   t1 = 10100 | 10111  ->  10111
Line 9:   t2 = 10100 ^ 10111  ->  00011
Line 10:  t3 = 10100 & 10010  ->  10000
Line 11:  t4 = 10111  100001 ->  110111
```

3.3 Data Transfer Instruction

There are two basic data transfer instructions for accessing data memory:

Code 3.3.1.

```
lw t0, 4(s3)   # load word from memory to register
sw t0, 8(s3)   # store word from register to memory
```

The data is loaded or stored using a 5-bit address. The memory address is formed by adding the contents of the base address register to the offset value.

Example.

```
.global _start

.data
a: .word 1 2 3 4 5

.text
_start:
        la a1, a
        lw t0, 0(a1)
        lw t1, 4(a1)
        lw t2, 8(a1)
        lw t3, 12(a1)
        lw t4, 16(a1)
        addi t4, t4, 1
        sw t4, 20(a1)
        lw t5, 20(a1)
```

```
t0 = 0x01, t1 = 0x02
t2 = 0x03, t3 = 0x04
t4 = 0x06, t5 = 0x06
```

> **Remark.** Address is byte-base, thus the increment is 4 when accessing `a1`.

These instructions follow the I-type format.

Since 8-bit bytes are useful, most architectures address individual bytes in memory.

Note that in byte addressing, we have Big Endian, where the leftmost byte is the word address, and the rightmost byte is the word address for Little Endian. In RISC-V, we use Little Endian, where the leftmost byte is the least significant byte.

We also have loading and storing byte operations:

Code 3.3.2.

```
lb t0, 1(s3)   # load byte from memory
sb t0, 6(s3)   # store byte to memory
```

Here, `lb` places the byte from memory into the rightmost 8 bits of the destination register and performs signed extension. `sb` then takes the byte from the rightmost 8 bits of a register and writes it to memory.

Example. Assume that in memory, we have:

```
0xFFFFFFFF    4
0x009012A0    0
```

Now, we have the following operation:

```
add s3, zero, zero
lb t0, 1(s3)
sb t0, 6(s3)
```

What is the value left in `t0`? What word is changed in memory and to what? What if the machine was Big Endian?

Solution:

1. `t0 =  0x00000012`
2. New memory:

```
0xFF12FFFF    4
0x009012A0    0
```

3. `t0 =  0x00000090`, New memory:

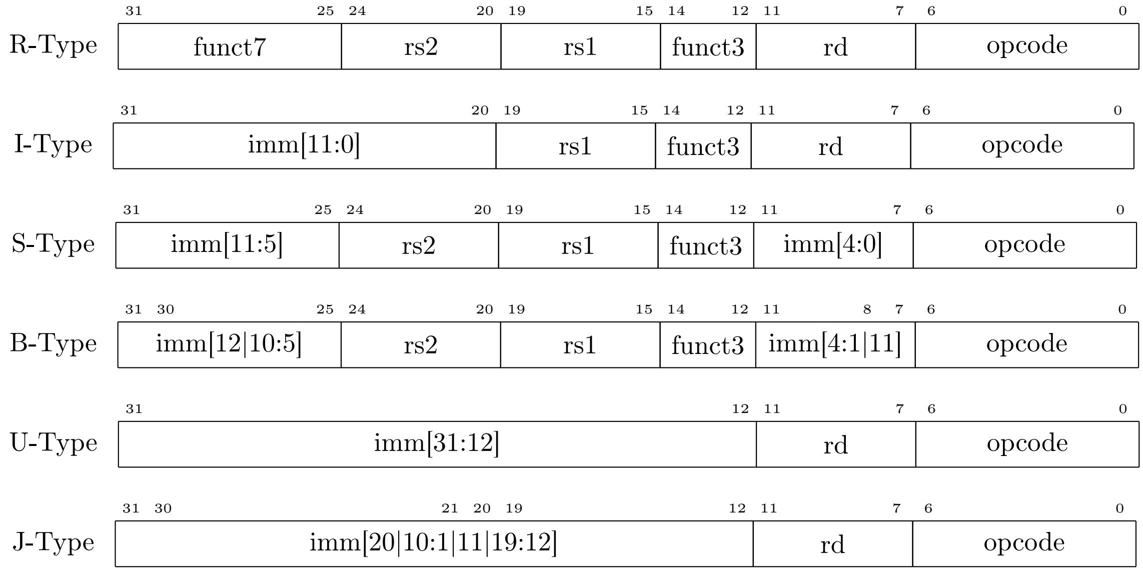
```
0xFFFF90FF    4
0x009012A0    0
```

Chapter 4

Control Instruction

4.1 Introduction to Register

Previously we have take a look on the instruction fields of RISC-V. Now, we can take a closer look on it.



There are a total of five instruction categories, including

1. Load and Store instruction
2. Bitwise instructions
3. Arithmetic instructions
4. Control transfer instructions
5. Pseudo instructions

The RISC-V register file holds 32 32-bit general-purpose registers, with two read ports and one write port. Thus, there are at most three operands. Registers are faster than main memory, and they are easier for the compiler to use. However, register files with more locations are slower.

4.2 Control Instructions

In RISC-V, we have control flow instructions. For example, we have conditional branch instructions:

Code 4.2.1.

```
bne s0, s1, Lbl   # go to Lbl if s0 != s1
beq s0, s1, Lbl   # go to Lbl if s0 == s1
```

These instructions follow the B-format.

Example.

```
1   .global _start
2
3   .text
4   _start:
5           li a0, 1
6           li a1, 1
7           li t0, 20
8           li t1, 23
9           bne t0, t1, inst1
10          addi a0, a0, 1
11          beq t0, t1, inst2
12          inst1:  addi a0, a0, 2
13          bne t0, zero, end
14          inst2:  addi a0, a0, 3
15          end:    sub a0, a0, a1
```

```
Line 5:   a0 = 1
Line 6:   a1 = 1
Line 7:   t0 = 20
Line 8:   t1 = 23
Line 9:   t0 != t1 -> goto inst1
Line 10 & 11 -> ignored
Line 12:  a0 = 3
Line 13:  t0 != 0  -> goto end
Line 14 -> ignored
Line 15:  a0 = 2
```

We need some extra instructions to support branch instructions. For example, we can use `slt` to support the branch-if-less-than instruction.

Code 4.2.2.

```
slt t0, s0, s1      # if s0 < s1, then t0 = 1; else, t0 = 0
slti t0, s0, 25     # if s0 < 25, then t0 = 1; else, t0 = 0 (signed)
sltu t0, s0, s1     # if s0 < s1, then t0 = 1; else, t0 = 0 (unsigned)
sltiu t0, s0, 25    # if s0 < 25, then t0 = 1; else, t0 = 0 (immediate unsigned)
```

This instruction follows R format or I format.

Example.

```
1   .global _start
2
3   .text
4   _start:
5           li a0, 1
6           li t0, 20
7           li t1, 23
8           slt a1, t0, t1
9           beq a0, a1, inst1
10          addi a0, a0, 2
11  inst1:  addi a0, a0, 3
```

```
Line 5:   a0 = 1
Line 6:   t0 = 20
Line 7:   t1 = 23
Line 8:   t0 < t1 -> a1 = 1
Line 9:   a0 == a1 -> goto inst1
Line 10:  ignored
Line 11:  a0 = 4
```

We can then use these instructions to create other conditions. We can also check for boundaries using these instructions. For example, with `slt` and `bne`, we can implement a branch-if-less-than:

Code 4.2.3.

```
slt t0, s1, s2        # t0 set to 1 if s1 < s2
bne t0, zero, Label
```

Treating signed numbers as if they were unsigned provides a low-cost way to perform these checks. For example:

Code 4.2.4.

```
sltu t0, s1, t2       # t0 = 0 if s1 > t2 (max)
                      # or s1 < 0 (min)
beq t0, zero, IOOB    # go to IOOB if t0 = 0
```

Since negative numbers in 2's complement look like very large numbers in unsigned notation, it checks both if `t0` is less than or equal to zero and greater than `t2`.

There are also unconditional branch instructions:

Code 4.2.5.

```
jal zero, Label    # go to Label, Label can be immediate value
j Label            # go to Label and discard return address
```

These instructions follow J format.

Example.

```
1   .global _start
2
3   .text
4   _start:                                   Line 5:   a0 = 1
5           li a0, 1                          Line 6:   t0 = 20
6           li t0, 20                         Line 7:   jump to Line 9
7           jal ra, loop                      Line 9:   a0 = 2, 3, ...
8   loop:                                     Line 10:  a0 != t0
9           addi a0, a0, 1                    Line 11:  keep looping
10          beq a0, t0, end                   Line 13:  a0 = 21
11          j loop
12  end:
13          addi a0, a0, 1
```

If the branch destination is further away than can be captured in 12 bits, we can use the following to perform a jump:

```
  bne s0, s1, L2
  j L1
L2: ...
```

Example. How a while-loop in C is compiled? For example

```
while (save[i] == k) i += 1;
```

Assume that `i` and `k` correspond to registers `s3` and `s5`, and the base of the array `save` is in `s6`.

Solution:

```
Loop: slli t1, s3, 2      # shift left 4 bytes (array operation)
      add t1, t1, s6      # t1 = address of save[i]
      lw t0, 0(t1)        # Temp reg t0 = save[i]
      bne t0, s5, Exit    # go to Exit if save[i] != k
      addi s3, s3,1       # i = i + 1
      j Loop              # go to Loop
Exit:
```

Remark. Left shifting `s3` is used to align the word address (4 bytes), and it is increased by 1 in `addi`. Thus, each time it is increased by 4.

Address of save[i] = save array address + shift address ($i \times 4$).

4.3 Accessing Procedures

Other than `jal`, we have branch instructions that return to the original location.

Code 4.3.1.

```
jal ra, label    # jump and link
jalr x0, 0(ra)   # return
```

Here, `jal` saves `PC + 4` by default into `ra`, so that when the procedure returns, it proceeds to the next instruction. `jalr` then uses the return address to return to the next procedure.

Example.

```
1   .global _start
2
3   .text
4   _start:                                 Line 5:   a0 = 20
5           li a0, 20                       Line 6:   a1 = 23
6           li a1, 23                       Line 7:   jump to Line 11
7           jal ra, add_two_numbers         Line 11:  a3 = 20
8           addi t1, a2, 0                  Line 12:  a4 = 23
9           j end                           Line 13:  a2 = 43
10  add_two_numbers:                        Line 14:  jump to Line 8
11          mv a3, a0                       Line 8:   t1 = 43
12          mv a4, a1                       Line 9:   jump to Line 16
13          add a2, a3, a4                  Line 16:  t1 = 44
14          jalr zero, 0(ra)
15  end:
16          addi t1, t1, 1
```

However, the number of registers is not enough for some operations. Thus, we use the stack, which is a last-in-first-out (LIFO) data structure. We use `sp` to address the stack, and it grows from high address to low address. To push data onto the stack, we use `sp = sp - 4`. To pop data from the stack, we use `sp = sp + 4`.

To allocate space on the stack, we have a frame pointer (`fp`) that points to the first word of the frame of a procedure, providing a stable base register for the procedure. `fp` is initialized using `sp` on a call, and `sp` is restored using `fp` on a return.

Example.

```
1   .global _start
2
3   .text
4   _start:
5           li a0, 20
6           li a1, 23
7           jal ra, add_two_numbers
8           addi t1, a2, 0
9           j end
10  add_two_numbers:
11          addi sp, sp, -8
12          sw a0, 4(sp)
13          sw a1, 0(sp)
14          add a2, a0, a1
15          lw a0, 4(sp)
16          lw a1, 0(sp)
17          addi sp, sp, 8
18          jalr zero, 0(ra)
19  end:
20          addi t1, t1, 1
```

```
Line 5:   a0 = 20
Line 6:   a1 = 23
Line 7:   jump to Line 11
Line 11:  assign 8 bytes in stack
          (from high to low)
Line 12:  save argument in stack 4(sp)
Line 13:  save argument in stack 0(sp)
Line 14:  a2 = 43
Line 15:  load argument from stack 4(sp)
Line 16:  load argument from stack 0(sp)
Line 17:  free stack
Line 18:  jump to Line 8
Line 8:   t1 = 43
Line 9:   jump to Line 16
Line 16:  t1 = 44
```

Example. Leaf procedures are ones that do not call other procedures. Give the RISC-V assembler code for the follows.

```
int leaf_ex (int g, int h, int i, int j)
{
  int f;
  f = (g + h) - (i + j)
  return f;
}
```

Solution: Suppose g, h, i, and j are in a0, a1, a2, a3:

```
leaf_ex:
        addi sp, sp, -8   # initialize stack room
        sw t1, 4(sp)      # save t1 on stack
        sw t0, 0(sp)      # save t0 on stack
        add t0, a0, a1
        add t1, a2, a3
        sub s0, t0, t1
        lw t0, 0(sp)      # restore t0
        lw t1, 4(sp)      # restore t1
        addi sp, sp, 8    # free stack
        jalr zero, 0(ra)
```

For nested procedures, we can store the return address on the stack so that, at the end, we can return to the original return address. For example, to find the factorial of a number, we can use:

Code 4.3.2.

```
fact:
      addi sp, sp, -8     # initialize stack pointer
      sw ra, 4(sp)        # save return address
      sw a0, 0(sp)        # save argument n
      slti t0, a0, 1      # test for n < 1
      beq t0, zero, L1    # if n >= 1, go to L1
      addi s0, zero, 1    # else return 1 in s0
      addi sp, sp, 8      # adjust stack pointer
      jalr zero, 0(ra)    # return to caller
L1:
      addi a0, a0, -1     # n >= 1, so decrement n
      jal ra, fact        # call fact with (n-1)
                          # this is where fact returns
bk_f:
      lw a0, 0(sp)        # restore argument n
      lw ra, 4(sp)        # restore return address
      addi sp, sp, 8      # free stack pointer
      mul s0, a0, s0      # s0 = n * fact(n-1)
      jalr zero, 0(ra)    # return to caller
```

Chapter 5

Logic basis

5.1 Numeral System

In common we use decimal, binary, octal and hexadecimal number systems. radix or base of the number system is the total number of digits allowed in the number system.

The conversion from a decimal integer to another number system is simple: divide the decimal number by the radix and save the remainder. Keep repeating the steps until the quotient is zero. The result is the reverse order of the remainders.

As shown in the previous chapter, we need to deal with signed integers. The original notation is simple, where we use the first bit of the binary string to represent the sign. For example, 1001_2 represents -1 and 0001_2 represents 1, which is called 1's complement. However, this leads to the situation where there are two types of zero: negative zero and positive zero.

Thus, we use 2's complement. We first complement all the bits and then add 1. For example, if we have -6 and want to represent it in binary notation, we have:

$$6_{10} = 0000\ 0000\ ...\ 0110_2 \Rightarrow 1111\ 1111\ ...\ 1001_2 + 1 \Rightarrow 1111\ 1111\ ...\ 1010 = -6$$

For an n-bit signed binary numeral system, the largest positive number is $2^{n-1} - 1$, and the smallest negative number is -2^{n-1}.

There are two types of signals: analog and digital. For an analog signal, it varies smoothly over time. For a digital signal, it maintains a constant level and then changes to another constant level at regular intervals. We can use 0 and 1 to represent a digital signal, with 1 being High/True/On/... and 0 being Low/False/Off/....

5.2 Logic Gates

Logic gates can produce different outputs for the same input signal. We can use a truth table to describe how the logic circuit's output depends on the logic levels of the inputs. For example, here is the truth table for an AND gate:

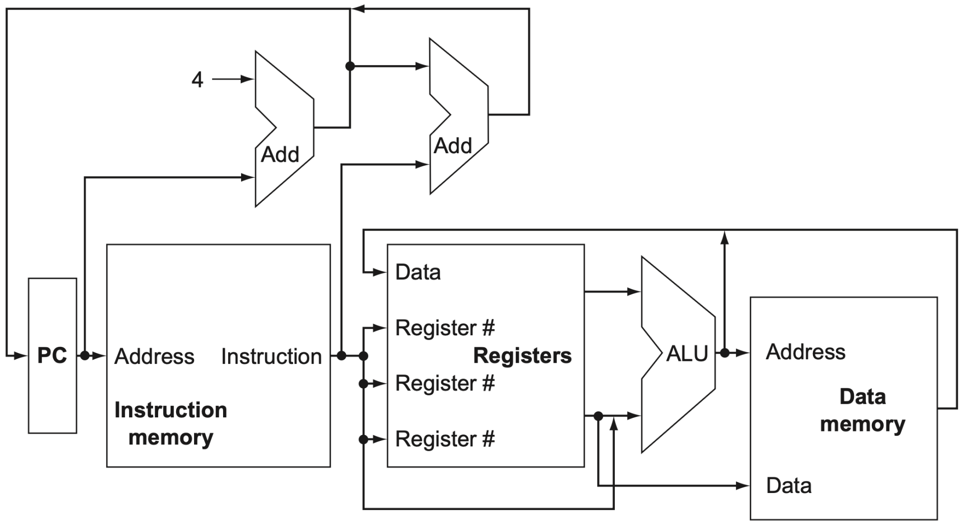
A	B	Output (A AND B)
0	0	0
0	1	0
1	0	0
1	1	1

Chapter 6

Arithmetic and Logic Unit

6.1 Overview

We can use the following to understand the abstract implementation:



Here, the ALU (Arithmetic Logic Unit) is responsible for performing arithmetic and logical operations. It receives instructions from the registers or instruction memory.

Before we dive into this topic, we can take a look on VHDL. VHDL is a hardware description language used to model and simulate the behavior of electronic systems, particularly digital circuits. It allows designers to describe the structure and functionality of a circuit at different levels of abstraction, from the behavioral to the structural level.

In the basic structure of VHDL, we design entity-architecture descriptions. The entity defines the system's interface, including externally visible characteristics such as ports and generic parameters. The architecture describes the system's internal behavior or structure, including internal signals and how the components interact. VHDL uses a time-based execution model to simulate and model the concurrent operations of digital systems.

For example, the assignment of A + B to result in the context of a Carry-Save Adder (CSA) would typically be part of the architecture description, as it defines the internal behavior and computation of the system.

For machine number representation, we use binary number integers. However, we need to consider storage limitations (overflow) and the representation of negative numbers.

In 32-bit signed numbers, the range is from $2^{31} - 1$ to -2^{31}. However, if the bit string represents an address, we only need to deal with unsigned integers, which range from 0 to $2^{32} - 1$.

To perform extension, we need to consider sign extension. Sign extension copies the most significant bit into the other bits to preserve the sign of the number. For example, to extend `0010`, we have `0000 0010`, and for `1010`, we have `1111 1010`.

Then, let's take a look at some arithmetic units.

6.2 Addition Unit

To build a 1-bit binary adder, we can use the XOR gate. Here's the truth table for the 1-bit adder:

A	B	Carry in	Carry out	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Where:

- $S = A \oplus B \oplus \text{Carry in}$

- Carry out $= (A\&B)|(A\&\text{Carry in})|(B\&\text{Carry in})$

To build a 32-bit adder, we can connect the carry-out of the least significant bit from the previous adder to the carry-in of the next least significant bit, and connect all 32 adders in sequence. This is called the Ripple Carry Adder. However, it is slow and involves a lot of glitching.

Glitching refers to the invalid and unpredictable output that can be read by the next stage, potentially resulting in incorrect behavior. This can be interpreted as a delay, where the outputs are not stable in time to be used in the subsequent operations.

The critical path (the longest sequence of dependent operations) is $n \times \text{CP}$, where n is the number of bits and CP is the time required for one full operation. This makes the Ripple Carry Adder slow because each bit's carry-out depends on the previous bit's carry-in, leading to a cumulative delay.

With the control unit, we can use the same structure to implement both an adder and a subtractor.

By tailoring the ALU, we can support various instructions in the ISA, including logic operations, branch operations, and others.

For example, after performing subtraction, we mark the result as 1 if the subtraction yields a negative result, and 0 otherwise. Then, we tie the most significant bit to the low-order bit of the input. This way, we complete a `slt` operation.

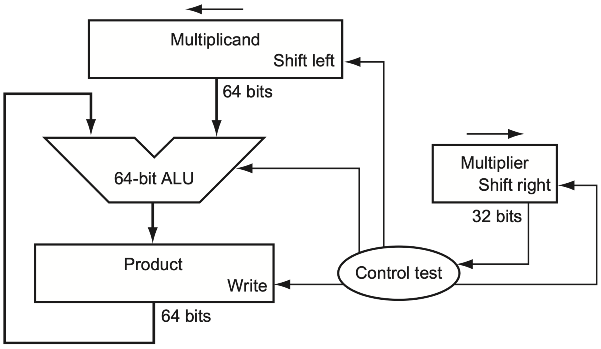
Overflow occurs when the result is too large to be represented. For example, adding two positive numbers yields a negative, adding two negative numbers gives a positive, subtracting a negative from a positive gives a negative, or subtracting a positive from a negative gives a positive. This leads to an exception. To fix this, we can modify the most significant bit to determine the overflow output setting.

6.3 Multiplication and Division

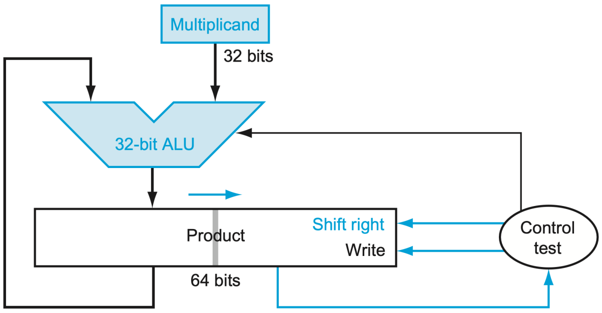
6.3.1 Multiplication

Multiplication is more complicated than addition. It can be accomplished by shifting and adding. For an n-bit × m-bit multiplication, we must have $n + m$ bits to cover all possible products.

The first version of multiplication needs a $2n$-bit adder for the multiplication of an n-bit and n-bit number, starting from the right half.



The refined version simplifies this by requiring only an n-bit adder for the same operation.



For example, when calculating $0010_2 \times 0011_2$, we have

0010 × 0011

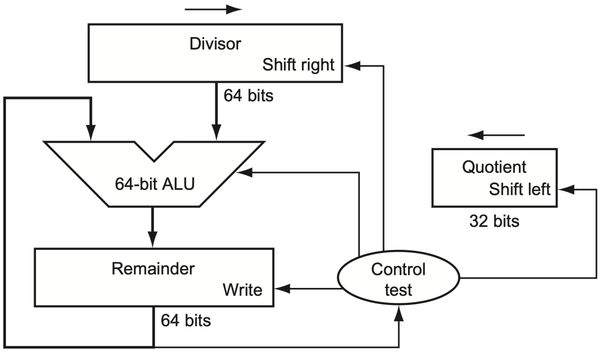
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001$\underline{1}$	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	000$\underline{1}$	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	000$\underline{0}$	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	000$\underline{0}$	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	000$\underline{0}$	0010 0000	0000 0110

`mul` performs a 32-bit × 32-bit multiplication and places the lower 32 bits in the destination register. `mulh`, `mulhu`, and `mulhsu` perform the same multiplication but return the upper 32 bits of the full 64-bit product.

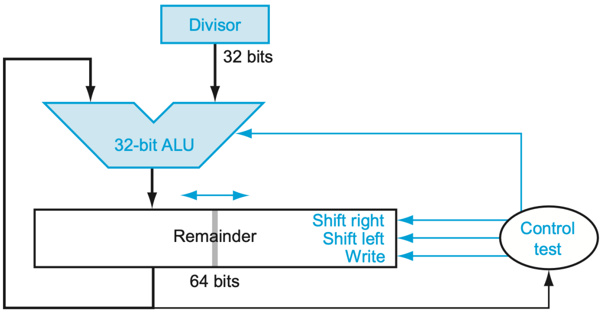
6.3.2 Division

Division is just a series of quotient digit guesses, left shifts, and subtractions.

In the first version of division, the 32-bit divisor starts in the left half of the divisor register and is shifted right 1 bit each iteration.



The refined version combines the Quotient register with the right half of the Remainder register.



`div` generates the remainder in `hi` and the quotient in `lo`. It performs a 32-bit by 32-bit signed integer division of `rs1` by `rs2`, rounding towards zero. `div` and `divu` perform signed and unsigned integer division of 32 bits by 32 bits. `rem` and `remu` provide the remainder of the corresponding division operation.

6.4 Shifter

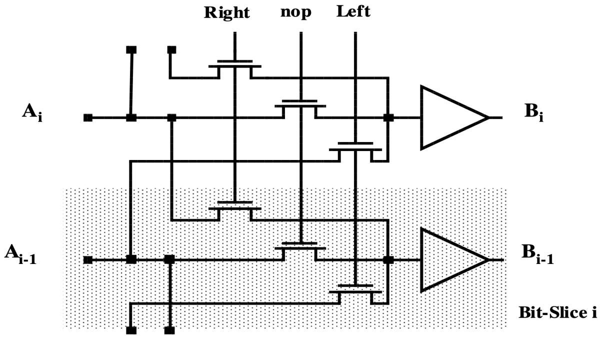
Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in rs1, and the shift amount is encoded in the lower 5 bits of the I-immediate field.

Code 6.4.1.

```
srli rd, rs1, imm[4:0]
srai rd, rs1, imm[4:0]
```

`slli` is a logical left shift, `srli` is a logical right shift, and `srai` is an arithmetic right shift. Logical shifts fill with zeros, while arithmetic right shifts fill with the sign bit. For example, a logical right shift of `1111` by 2 bits results in `0011`, while an arithmetic right shift of `1111` by 2 bits results in `1111`.

A simple shifter can be accomplished by using a series of multiplexers to shift the input data by a specified number of bit positions, either left or right.



For example, to do a right shift, let Right $= 1$ and nop $=$ Left $= 0$. Then, $B_{i-1} = A_i$, where B is the shifted output and A is the input.

In a parallel programmable shifter, we can use control signals to decide the shift amount, direction, and type. The control logic determines how many positions the data should be shifted, whether it should be shifted left or right, and whether the shift should be logical or arithmetic. This allows for flexible shifting operations based on the input values and the specified parameters.

A logarithmic shifter is a more complex shifter that can perform shifts based on logarithmic scaling. It involves specialized shifting mechanisms used for fast multiplication and division by powers of 2. With one shifter, we can perform a shift by 0 or 1 bit; with two shifters, we can perform shifts by 0, 1, 2, or 3 bits, and so on.

Chapter 7

Floating Numbers

We discussed the representation of integers in previous chapters, and the representation of floating-point numbers is more complex.

However, we can break a floating-point number into parts. For example, consider

$$\underbrace{6.6254}_{\text{Mantissa (always positive)}} \times \underbrace{10}_{\text{Base}}{}^{\overbrace{-27}^{\text{Exponent}}}$$

We have:

1. Mantissa: A normalized number with a certain level of accuracy (e.g., 6.6254).
2. Exponent: A scale factor that determines the position of the decimal point (e.g., 10^{-27}).
3. Sign bit: Indicates whether the number is positive or negative.

We normalize the mantissa to fall within the range $[1, R)$, where R is the base. For instance, in the case of a binary base, this range would be $[1, 2)$.

In IEEE Standard 754 Single Precision, we have

S	E'	M

Here, S represents the sign bit, where 0 indicates a positive number and 1 indicates a negative number. E' is the 8-bit signed exponent, represented in excess-127 notation, ranging from -127 to 128. M is the 23-bit mantissa fraction. The value is thus represented as $\pm 1.M \times 2^{E'-127}$.

> **Remark.** Minimum exponent $= 1 - 127 = -126$; Maximum exponent $= 254 - 127 = 127$

For double precision, we use 64 bits. E' is the 11-bit signed exponent, represented in excess-1023 notation, and M is the 52-bit mantissa fraction.

> **Example.** What is the IEEE single precision number $\texttt{40C0000}_{16}$ in decimal?
>
> **Solution:** First, convert the hexadecimal number $\texttt{40C0000}_{16}$ to binary:
>
> $$\texttt{40C0000}_{16} = \texttt{0100 0000 1100 0000 0000 0000 0000 0000}_2$$
>
> Sign bit (0): Positive (+)
>
> Exponent: $\texttt{10000001}_2 - 127 = 129 - 127 = 2$
>
> Mantissa: $\texttt{1.100 0000 0000 0000 0000 0000}_2 = 1 + 1 \times 2^{-1} = 1.5$
>
> Therefore, the result is:
>
> $$1.5 \times 2^2 = 6_{10}$$

Example. What is -0.5_{10} in IEEE single precision binary floating-point format?

Solution: Sign bit: 1 (since the number is negative)

Mantissa: $0.5_{10} = 1.0 \times 2^{-1} = 0.1_{2c}$

Exponent: $127 - 1 = 126 = 01111110$

Thus, in binary:

$$-0.5_{10} = \texttt{1011 1111 0000 0000 0000 0000 0000 0000}_2$$

Remark. Exponents with all 0's or all 1's have special meanings in IEEE floating-point representation:

- $E = 0, M = 0$: Represents 0.

- $E = 0, M \neq 0$: Represents a denormalized number, which is $\pm 0.M \times 2^{-126}$.

- $E = 1 \ldots 1$, $M = 0$: Represents $\pm\infty$, depending on the sign.

- $E = 1 \ldots 1$, $M \neq 0$: Represents `NaN` (Not a Number).

Chapter 8

Datapath

Now we can take a closer look at the implementation of RISC-V.

8.1 Overview

In the implementation, we use the program counter (PC). After supplying the instruction address and fetching the instruction from memory, we update the PC. Then, we decode the instruction and execute it.

There are two types of functional units (logic elements). The first type is combinational (combinational elements), which operate on data values. The output of these functional units depends only on the current input, meaning there is no internal storage. The second type includes elements that contain state, called state elements. These elements have internal storage, which characterizes a computer. For example, instruction memory, register files, and data memory are sequential (state elements), while the ALU is combinational.

> **Remark.** In instruction memory, instructions are placed one by one. In register files, there are 32 lines, and we only need 5 bits in the instruction to indicate the register file address.

To fetch instructions, the processor first reads the instructions from the instruction memory, then updates the PC to the address of the next instruction. The PC is updated every clock cycle (typically PC = PC+4 by default), and the instruction memory is read every clock cycle.

Note that the clock is edge-triggered, so the PC is updated only on the rising or falling edge of the clock.

After the instructions are read, the processor decodes them. The fetched instruction's opcode and function field bits are sent to the control unit, which then generates control signals used in the future datapath. Next, two values are read from the Register File, with the register addresses contained in the instruction.

Regardless of whether the values are actually used, the Register File's read ports are active for all instructions during the decode cycle. In case the instruction requires values from the Register File, it reads the two source operands by default.

All instructions, except `j`, use the ALU after reading from the register. For memory-reference instructions, the ALU is used to compute addresses; for arithmetic instructions, the ALU performs the required arithmetic; for control instructions, the ALU computes branch conditions.

8.2 Operations

8.2.1 R Format Operations

	31 25	24 20	19 15	14 12	11 7	6 0
R-Type	funct7	rs2	rs1	funct3	rd	opcode

R-type instructions perform operations on values in `rs1` and `rs2`, then store the result back into the Register File. Note that the Register File is not written every cycle, so a write control signal is needed for the Register File.

8.2.2 I and S Format Operations

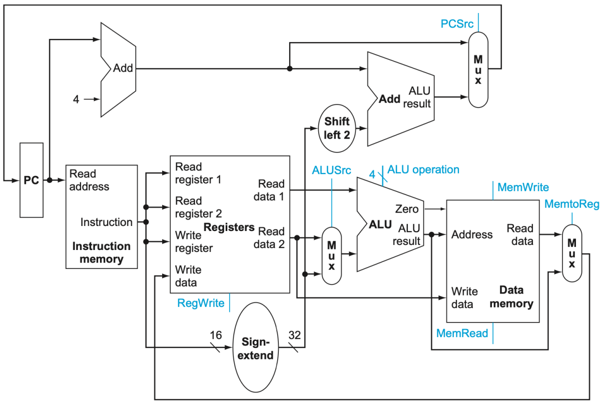
	31 20	19 15	14 12	11 7	6 0
I-Type	imm[11:0]	rs1	funct3	rd	opcode

	31 25	24 20	19 15	14 12	11 7	6 0
S-Type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

For load and store operations, the memory address is computed by adding the base register to the 12-bit signed offset field in the instruction (`imm[] + rs1`). The base register is read from the Register File during decode, and the offset value in the lower 12 bits of the instruction must be sign-extended to create a 32-bit signed value.

For store instructions, the value is read from the Register File during decode and written into the Data Memory. For load instructions, the value is read from the Data Memory and then stored into the Register File.

Also, note that the `lw` and `sw` instructions access data memory, not instruction memory.



As shown above, after decoding, the sign is extended in the lower part. Above is the clock for the PC value, which executes the branch instruction. In the rightmost mux, it selects the source. It is activated only for `lw` instructions. Additionally, only for `sw`, the MemWrite signal will be 1 (High), which activates the write data port.

8.2.3 B Format Operations

	31 30 25	24 20	19 15	14 12	11 8	7 6 0
B-Type	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode

For branch operations, it compares the operands read from the Register File during decode for equality. The 12-bit B-immediate encodes signed offsets in multiples of one byte. It is sign-extended and added to the address of the branch instruction to give the target address.

8.2.4 J Format Operations

	31 30 21 20 19 12	11 7	6 0
J-Type	imm[20|10:1|11|19:12]	rd	opcode

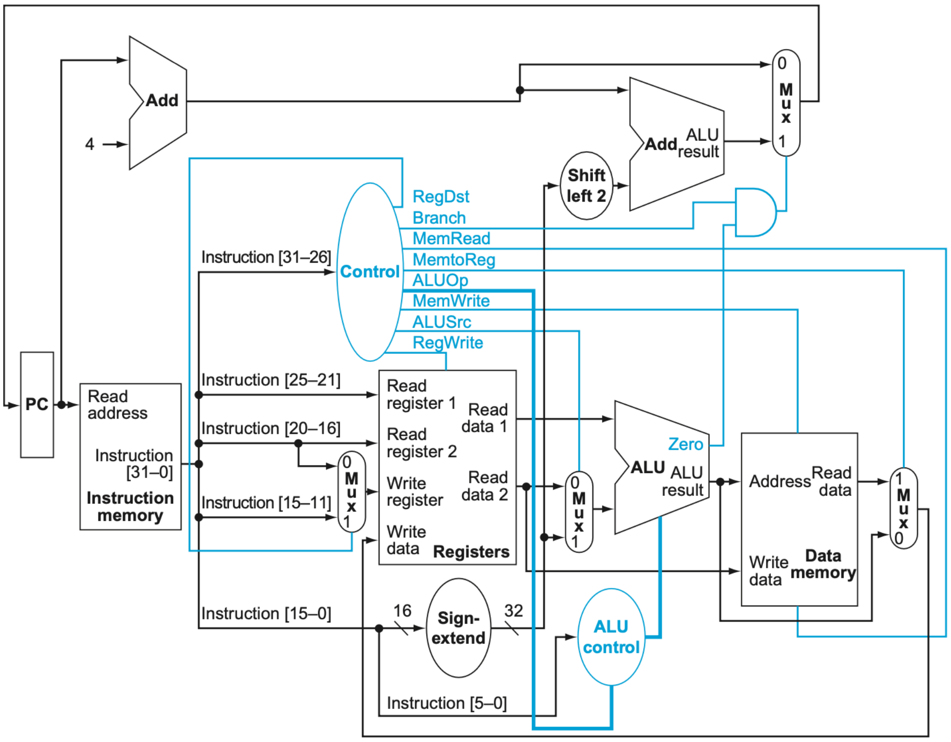
The J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Since `imm[0]` is always 0, we don't have it in the instruction.

The partition of `imm` field is designed to align the `imm` bits better with other instruction types, enabling a more efficient implementation of control units.

8.3 Datapath

By assembling the above datapath elements, adding control lines, and designing the control path, we can create a single datapath.

We need to fetch, decode, and execute each instruction in one clock cycle, which is called the single-cycle design. No datapath resource can be used more than once per instruction, so some components must be duplicated. Additionally, multiplexers are needed at the input of the shared elements with control lines to perform the selection, allowing datapath elements to be shared between different instructions.



Here, the MUX before the ALU determines the second ALU operand, and the MUX after the Data Memory decides whether to feed memory data to the register file. The system clock is edge-triggered and is determined by the length of the longest path. The ALU is used to compute the branch instruction, and its output can replace the PC when needed.

Memory and Register File reads are combinational. By using write signals along with the clock edge, we determine when to write to the sequential elements, such as the PC or Register File.

By adding the control as shown above, we can select the operations to perform, control the flow of data, and direct the information that comes from the 32-bit instruction.

> **Remark.** The instruction is decoded in the path between the Instruction Memory and Register File.

For different operations, the control signals vary.

	`add`	`lw`	`sw`	`beq`
MUX after Reg	0	1	1	0
MUX after DataMem	0	1	/	/
MUX after Add	0	0	0	/
RegWrite	1	1	0	0
MemWrite	0	0	1	0
MemRead	0	1	0	0

When the MUX after Add = 0, the PC is updated as PC+ = 4. Both ALU inputs are from the Register File.

Chapter 9

Pipeline

9.1 Motivations

As discussed before, an instruction finishes within a single clock cycle. However, this can be inefficient since the clock cycle must be timed to accommodate the slowest instruction, meaning every instruction takes the same amount of time. This results in wasted area, as some functional units (e.g., adders) must be duplicated since they cannot be shared within a single clock cycle. Additionally, for simple instructions, the latter part of the clock cycle might be wasted.

Example. Calculate the cycle time assuming negligible delays (for muxes, control unit, sign extension, PC access, shift left by 2, wires) except for:

- Instruction fetch and update PC (IF), read/write data from/to data memory (MEM) (4 ns)

- Execute R-type; calculate memory address (EXE) (2 ns)

- Register fetch and instruction decode (ID), write the result data into the register file (WB) (1 ns)

Solution:

Instruction	IF	ID	EXE	MEM	WB	Total
R / I type	4	1	2		1	8
`lw`	4	1	2	4	1	12
`sw`	4	1	2	4		11
`beq`	4	1	2			7
`jal`	4	1	2		1	8
`jalr`	4	1	2		1	9

Therefore, we try to make it faster by fetching and executing the next instructions while the current instruction is running, and we introduce the concept of pipelining here. Under ideal conditions, with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipeline stages. For example, a five-stage pipeline is nearly five times faster because the clock cycle is "nearly" five times faster.

Also, we have

$$\text{CPU time} = \text{CPI} \times \text{CC} \times \text{IC},$$

where CPI = cycles per instruction, CC = clock cycle time, and IC = instruction count.

By pipelining, it reduces the time spent on each clock cycle and decreases the CPU time.

9.2 Pipeline Basis

Instructions are divided into five stages:

- IF: Instruction fetch and PC update

- ID: Instruction decode and register file read

- EXE: Execution or address calculation

- MEM: Data memory access

- WB: Write the result data back into the register file

By dividing the stages, we can increase the total amount of work done in a given time. However, instruction latency, which is the time from the start of an instruction to its completion, is not reduced.

Similarly, the clock cycle is limited by the slowest stage, so some stages do not need the whole clock cycle.

This might lead to the situation where, for example, if we have IF = 100ps, ID = 100ps, EXE = 200ps, MEM = 200ps, and WB = 100ps, the latency of an instruction takes 1000ps in a pipelined case, while it takes 700ps in a non-pipelined case. However, for more instructions, the overall speed is faster in the pipelined case than in the non-pipelined case.

In RISC-V, the implementation of the pipeline is relatively simple for the following reasons:

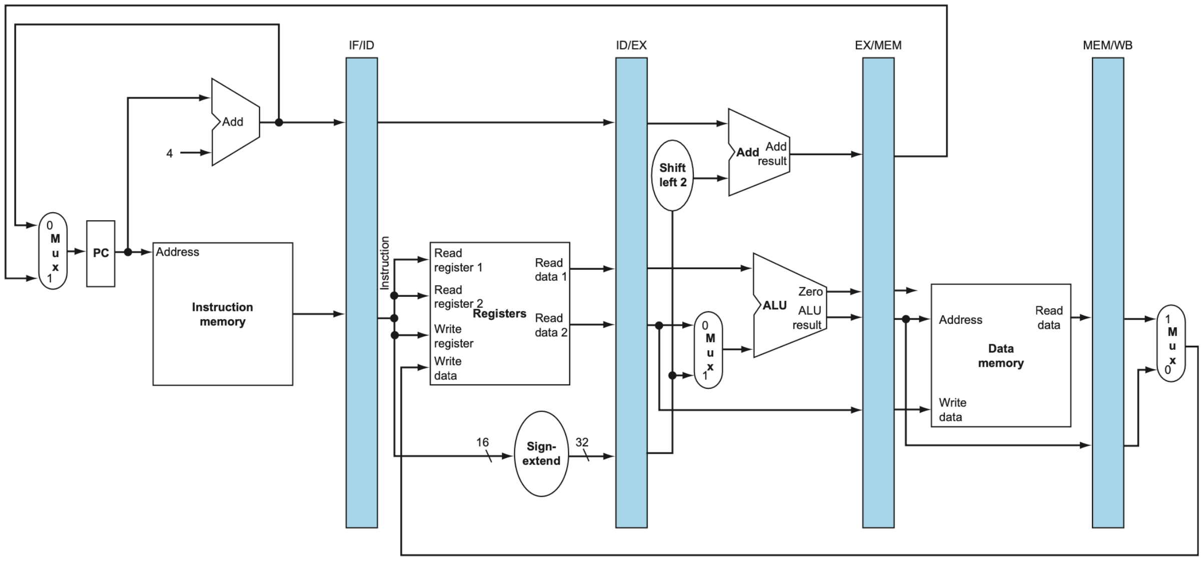
1. All instructions have the same length.

2. There are few instruction formats with symmetry across formats.

3. Memory operations occur only in loads and stores.

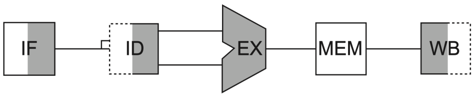
4. Each instruction writes at most one result, and it does so in the last few pipeline stages.

5. Operands must be aligned in memory, so a single data transfer takes only one data memory access, which is accomplished in RISC-V fields.



State registers are placed between each pipeline stage to isolate them. Each register is a flip-flop, and data moves in at the rising edge. After the pipeline is fully utilized, we can complete one instruction per cycle.

To simplify, we use graphics to represent the RISC-V pipeline.



Other pipeline structures are also possible.

We use pipelines because they are better for performance. Once the pipeline is full, one instruction is completed per cycle, so the CPI (cycles per instruction) is 1.

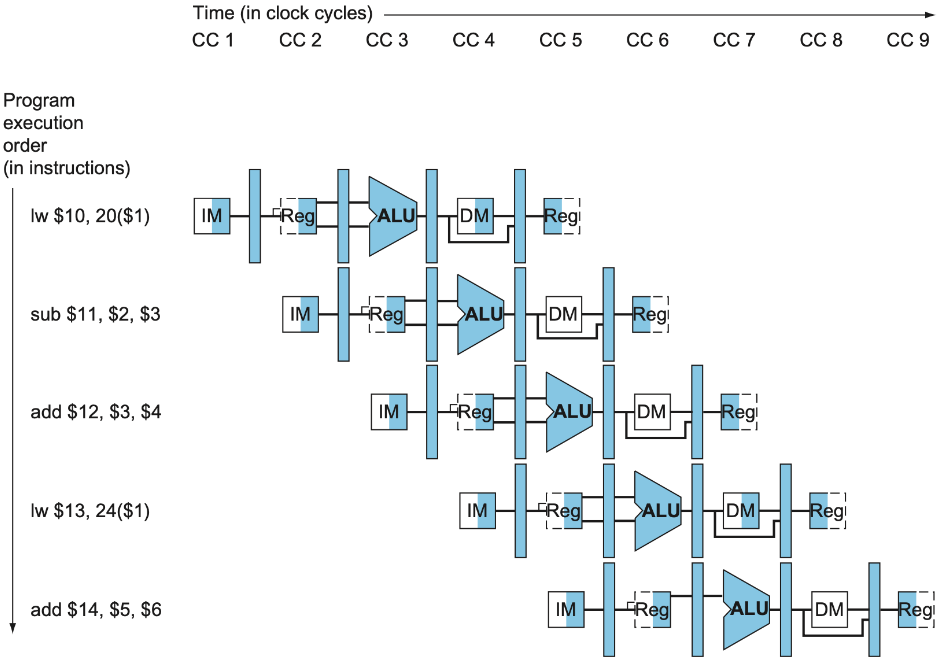
However, pipelines can cause issues, as they may introduce hazards. There are three possible pipeline hazards: structural hazards, caused by a busy resource; data hazards, where data is attempted to be used before it is ready; and control hazards, where control actions depend on the outcome of a previous instruction.

We typically resolve these hazards by allowing pipeline control to detect the hazards and take action to resolve them.

9.3 Structural Hazards

Structural hazards are caused by conflicts in the use of a resource. In a RISC-V pipeline with a single memory, it needs to access both data and instructions to load or store data and fetch new instructions. Therefore, the pipeline datapaths require separate instruction and data memories to avoid such conflicts.

To resolve a structural hazard, we can provide additional hardware components.



As mentioned above, by separating instruction and data memories, we can resolve the structural hazard. For example, in the diagram above, while the first `lw` is reading data from memory, the second `lw` is reading instructions from memory. Since the memories are separated, the issue is resolved.

In the diagram above, `sub` and the second `add` instructions are accessing the same register file, which could lead to a structural hazard. This can be fixed by performing reads in the second half of the cycle and writes in the first half. We use the clock edge to control the register writing and loading.

9.4 Clocking Methodology

Clocking methodology defines when signals can be read and when they can be written. The clock rate is given by:

$$\text{Clock rate} = \frac{1}{\text{Clock cycle time}}$$

This can be implemented using level-sensitive latches, master-slave flip-flops, or edge-triggered flip-flops.

The change of state is based on the clock. For latches, the output changes whenever the inputs change and the clock is asserted (level-sensitive methodology). For flip-flops, the output changes only on a clock edge (edge-triggered methodology).

Chapter 10

More on Pipeline

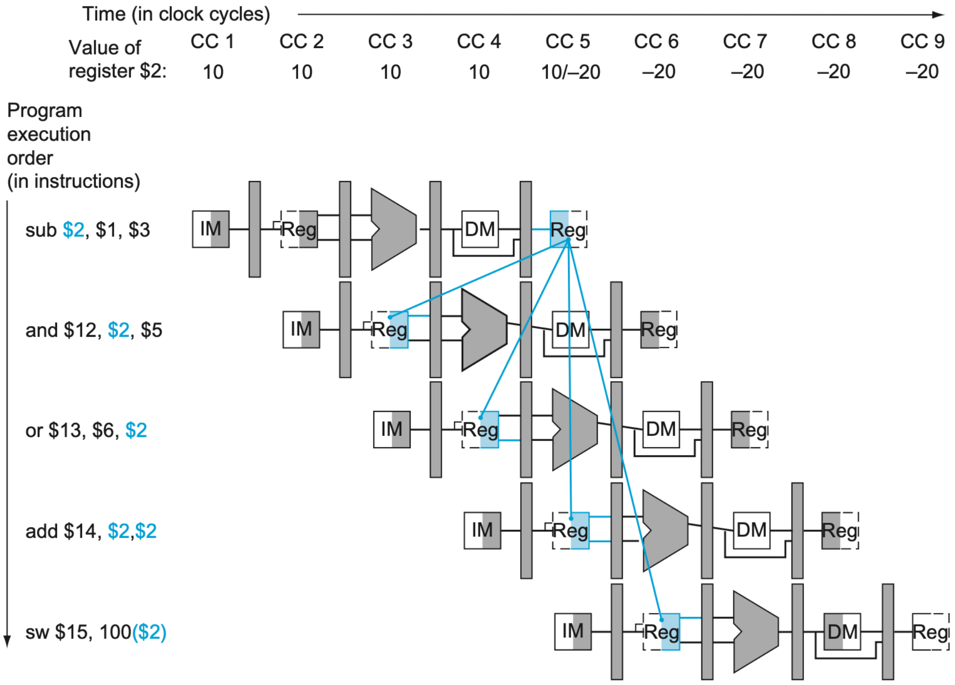
Here we continue the discussion on pipeline hazards.

10.1 Data Hazards

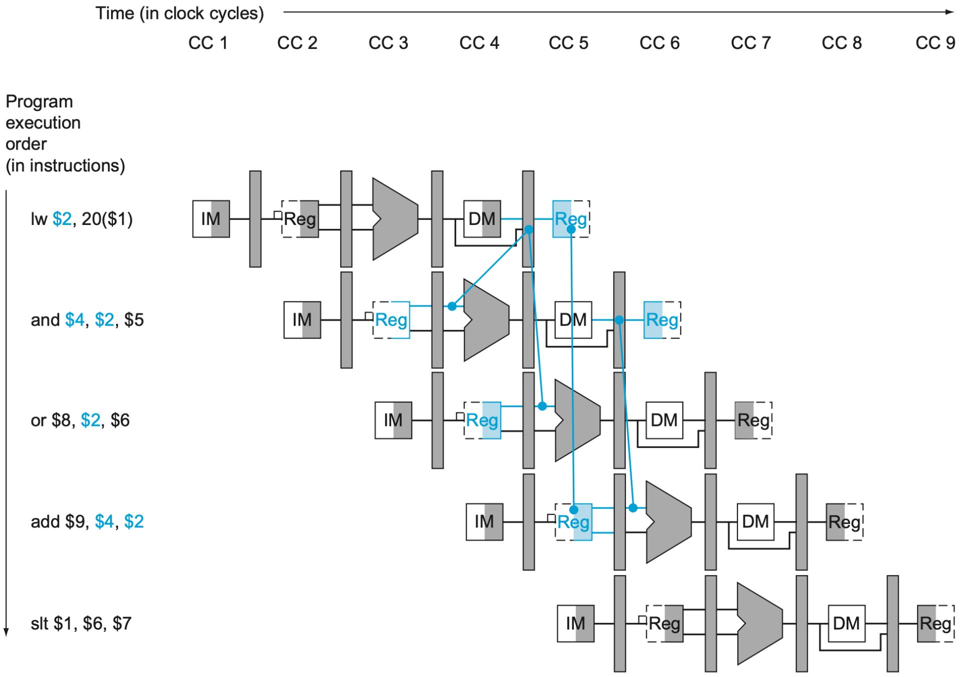
Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. To put it simply, this happens because a previous step has not finished data manipulation, while a following step requires the result from the previous step. This leads to data hazards.

Data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline. There are two main types of data hazards:

The first type is the Read After Write (RAW) data hazard. This occurs when an instruction needs to read a value that has not yet been written by a previous instruction. For example, as shown below, the `add` instruction requires a value that is produced by the preceding `sub` instruction, but `sub` will only write the result to the register during its final pipeline stage.



The second type is the Load-Use data hazard. This occurs when an instruction needs to use data that is being loaded from memory by a previous instruction, but the load (`lw`) operation will only complete at the final pipeline stage. As shown below, the `and` instruction depends on the result of the `lw` instruction, leading to a data hazard.



To solve this problem, there are two main methods:

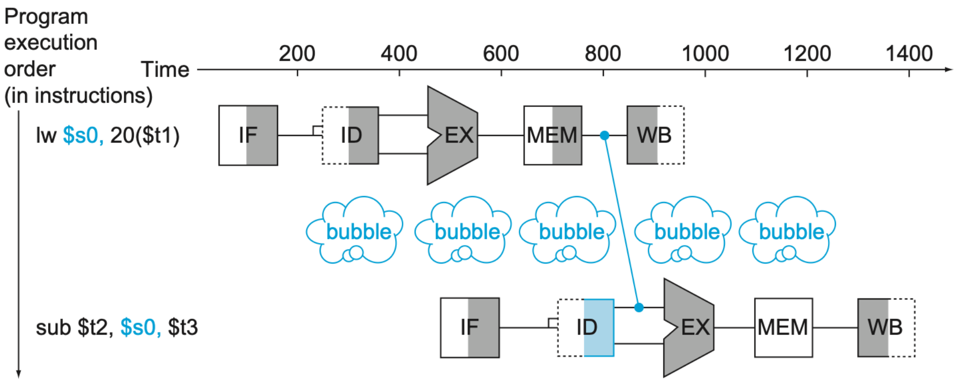
1. Insert NOP (no operation) / Stall

One method is to insert NOPs or stall by waiting for the preceding instruction to complete. This can resolve the hazard, but it reduces the overall CPI (Cycles Per Instruction), which is not desirable.

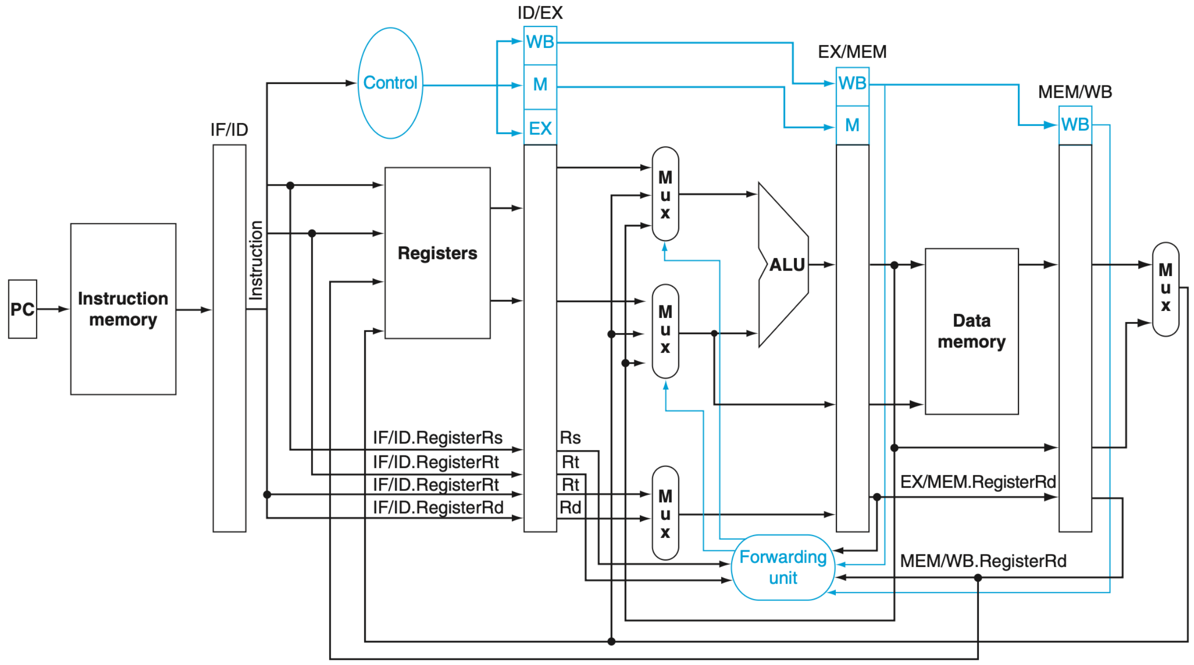
2. Forwarding

The second method is comparatively better than the first. We can resolve data hazards by forwarding results as soon as they are available to where they are needed.

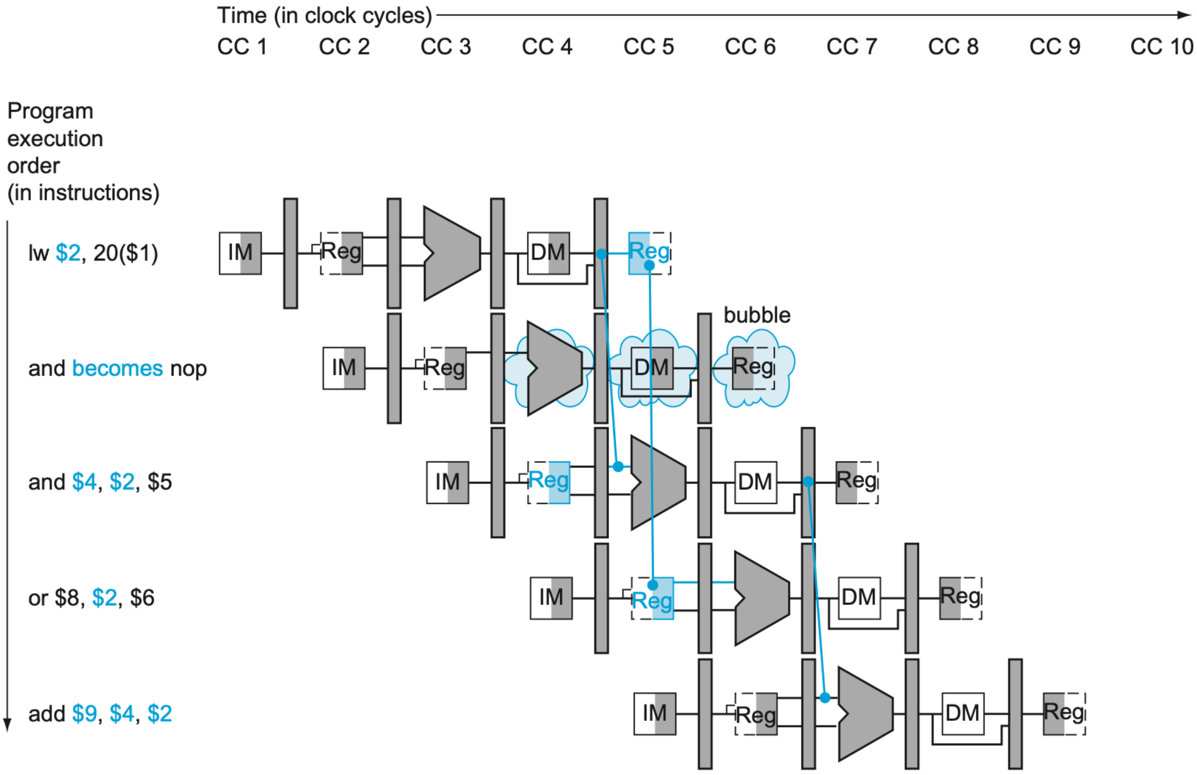
However, sometimes a combination of both methods is required. As shown below, after the `lw` instruction, we need to use both stalling and data forwarding to ensure that the data is transferred correctly.



Then, we can modify the datapath by adding forwarding hardware to handle data hazards more efficiently.



As mentioned before, the combination of stalling and forwarding is required for some instructions. Here is another demonstration:



As shown in the diagram, data will only be written to the register at the REG stage. We cannot do data forwarding directly, since the register read process occurs earlier than the register write in the pipeline. Thus, we need to stall one instruction. After that, we can perform data forwarding directly.

Note that we prefer the data from the newer instruction; thus, it is sometimes necessary to add a stall to avoid forwarding the outdated data.

For the or instruction, it works fine since the register reading and writing are done simultaneously.

10.2 Control Hazards

The next type is the control hazard. Control hazards occur when the flow of instruction addresses is not sequential, arising from the need to make a decision based on the results of one instruction while others are still executing. They happen due to changes in the flow of instructions, such as:

1. Unconditional branches: `jal`, `jalr`
2. Conditional branches: `beq`, `bne`
3. Exceptions

Again, to put it simply, this happens because we cannot determine whether to proceed with the next instruction after a branch instruction. This is due to the fact that the branch condition may or may not be satisfied, potentially resulting in a jump to a different instruction.

However, control hazards occur less frequently than data hazards, and there is nothing as effective against control hazards as forwarding is against data hazards. Therefore, we will take a look on a few possible strategies.

Control hazards can be mitigated by stalling, moving the decision point as early as possible in the pipeline, delaying the decision, or using prediction techniques.

10.2.1 Pipeline Stalls

First, we can take a look at how control hazards occur.

In jump instructions, the instruction is not decoded until the ID stage. This means that the pipeline might execute sequential instructions before determining whether the jump is taken, causing incorrect instructions to be processed due to the pipeline's nature. This results in a control hazard, and it's undesirable because the pipeline has already committed to the wrong instructions. We can fix this issue by using a single stall.

Fortunately, jumps are relatively infrequent, reducing their impact on performance.

However, branches introduce a more complicated case.

For branch instructions, control hazards occur due to the need to resolve the condition (e.g., whether the branch is taken or not) before proceeding with the correct instructions. This delay in knowing the outcome of the branch results in pipeline hazards, often requiring mechanisms like branch prediction to mitigate the impact.

To solve this problem, we can again use stalls. By flushing three instructions, we can easily fix the issue. However, as with data hazards, this still affects the overall CPI.

> **Remark.** There are two types of stalls. A **NOP** instruction is inserted between two instructions in the pipeline. It prevents the instructions earlier in the pipeline from progressing down the pipeline for a cycle. A **flush** discard instructions in a pipeline, usually due to an unexpected event. The key difference is that a `NOP` is determined at the compilation stage, while a flush is determined during runtime.

10.2.2 Delayed Branching

Another method to solve this problem is by moving the branch decision hardware as early in the pipeline as possible, i.e., during the decode (`ID`) cycle. This helps reduce the delay caused by branch instructions.

To reduce stalls, we can move the branch decision hardware back to the execute (`EX`) stage, which reduces the number of stall cycles to two.

Additionally, we can add hardware to compute the branch target address and evaluate the branch decision during the `ID` stage. This will reduce the number of stalls to one. The branch target computation can be done in parallel with register file read.

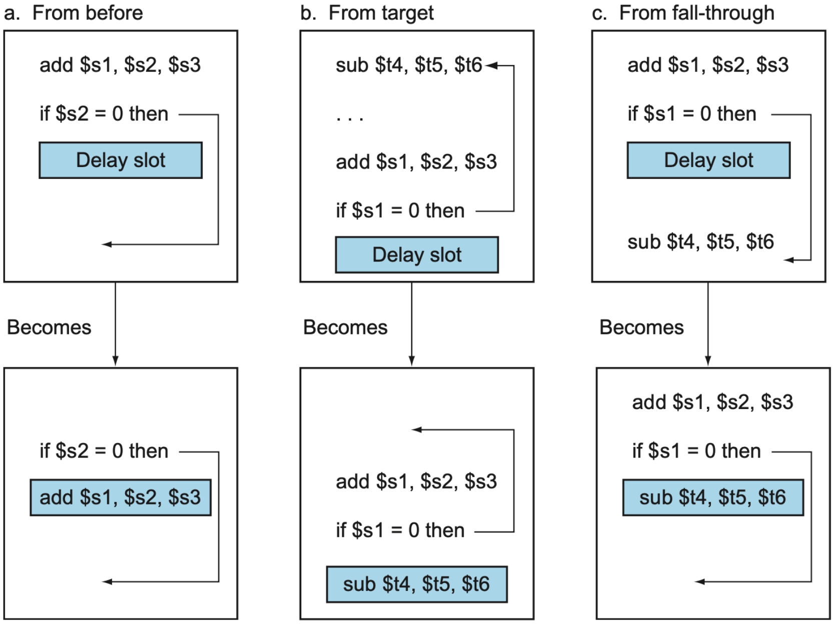
Notice that if there is an instruction right before a branch that produces one of the branch source operands, then a stall must be inserted. This is because the `EX` stage ALU operation is occurring at the same time as the `ID` stage branch compare operation.

If the branch hardware is moved to the `ID` stage, we can eliminate all branch stalls using delayed branches. Delayed branches are defined as always executing the next sequential instruction after the branch instruction — the branch takes effect only after that instruction.

To achieve this, the compiler moves an instruction that is not affected by the branch (a safe instruction) immediately after the branch, thereby hiding the branch delay. With deeper pipelines, the branch delay increases, potentially requiring more than one delay slot.

For the compiler or software, we can schedule the branch delay slots as shown below.

Here, (a) is the best choice, as it fills the delay slot with an instruction, which reduces the instruction count (IC). In (b) and (c), the subtraction instruction may need to be duplicated, increasing the IC. However, they still must be able to execute the subtraction if the branch is not taken.



> **Remark.** A **branch delay slot** is the slot directly after a delayed branch instruction. In the MIPS architecture, it is filled by an instruction that does not affect the branch.

10.2.3 Branch Prediction

We can also use **branch prediction** to resolve the branch delay by assuming a given outcome and proceeding without waiting for the actual branch result. Although branch prediction is more expensive than using delayed branching, the increasing number of available transistors has made it a more feasible and cost-effective solution in modern processors.

Static Branch Prediction

One prediction we can make is *branch not taken*. In this case, we always predict that branches will not be taken and continue fetching from the sequential instruction stream. Only when the branch is taken does the pipeline stall. If it is taken (the prediction is wrong), we flush the instructions after the branch and restart the pipeline at the branch destination.

Predict not taken works well for *top of the loop* branching structures, but such loops have jumps at the bottom to return to the top, incurring the jump stall overhead. Additionally, prediction *not taken* does not work well for *bottom of the loop* (do-while loop) branching structures.

Another prediction we could make is *branch taken*. This always incurs one stall cycle. As the branch penalty increases, a simple static prediction scheme will hurt performance. With more hardware, it is possible to predict branch behavior dynamically during program execution.

Dynamic Branch Prediction

We can also use dynamic branch prediction, where we predict branches at run-time using run-time information. This uses historical information to make more accurate predictions.

A branch prediction buffer (also known as a Branch History Table (BHT)) in the IF stage is addressed by the lower bits of the PC. It contains a bit (or bits) passed to the ID stage through the IF/ID pipeline register, indicating whether the branch was taken the last time it was executed.

The prediction bit may be incorrect (it could be a wrong prediction for this branch iteration or from a different branch with the same low-order PC bits), but this does not affect correctness, only performance.

The branch decision occurs in the ID stage after determining that the fetched instruction is a branch and checking the prediction bit(s).

If the prediction is wrong, we flush the incorrect instruction(s) in the pipeline, restart the pipeline with the correct instruction, and invert the prediction bit(s).

If the prediction is correct, stalls can be avoided no matter which direction they go.

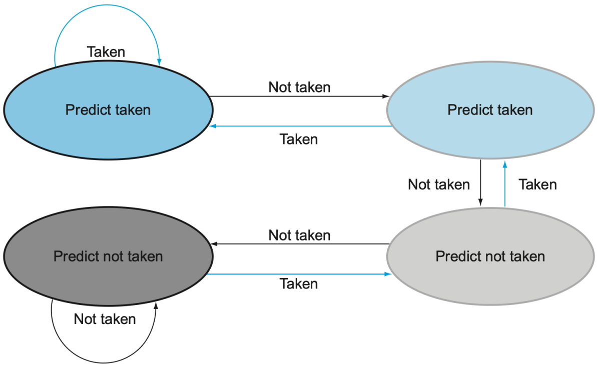
We also need a **Branch Target Buffer (BTB)**, which caches the target addresses of previously executed branch instructions.

We have two types of predictor.

A 1-bit predictor will be incorrect twice when not taken:

- Assume `predict bit` = 0 to start (branch not taken) and loop control is at the bottom of the loop code.
- The first time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop. The prediction bit is inverted (`predict bit` = 1).
- As long as the branch is taken (looping), the prediction is correct.
- Exiting the loop, the predictor again mispredicts the branch, since this time the branch is not taken (falling out of the loop). The prediction bit is inverted (`predict bit` = 0).
- For 10 times through the loop, we have an 80% prediction accuracy for a branch that is taken 90% of the time.

A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed.



10.3 Exceptions

Exceptions (also known as interrupts) are just another form of control hazard. Exceptions arise from:

- R-type arithmetic overflow
- Trying to execute an undefined instruction
- An I/O device request
- An OS service request (e.g., a page fault, TLB exception)
- A hardware malfunction

The pipeline has to stop executing the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address (the address of the exception handler code). The software (OS) looks at the cause of the exception and deals with it.

There are two types of exceptions. The first one is Interrupts, which are asynchronous to program execution. These are caused by external events and may be handled between instructions, allowing the instructions currently active in the pipeline to complete before passing control to the OS interrupt handler. We simply suspend and resume the user program.

The second type is Traps, which are synchronous to program execution. These are caused by internal events. The condition must be remedied by the trap handler for that instruction, so the offending instruction must stop midstream in the pipeline and pass control to the OS trap handler. The offending instruction may be retired, and the program may continue, or it may be aborted.

To put it simply, it's just a bug occurring in the code, and the compiler aborts. Traps are exceptions like division by zero, accessing invalid memory, etc., which are called synchronous. Interrupts are external events, like network devices needing attention, or pressing keyboard while compiling.

Notice that multiple exceptions can occur simultaneously in a single clock cycle.

In pipelined execution, arithmetic overflow (in the `EX` stage), undefined instruction (in the `ID` stage), and page fault (in the `IF` or `MEM` stage) are examples of **synchronous exceptions**, while I/O service requests and hardware malfunctions can occur in any stage and are classified as **asynchronous exceptions**.

Chapter 11

Performance

11.1 Performance

The goal of understanding performance is to identify the factors in the architecture that contribute to overall system performance, as well as the relative importance (and cost) of these factors.

Here, we consider two performance metrics.

The first is **Response Time** (execution time), which is the time between the start and completion of a task. This metric is important to individual users. The second is **Throughput** (bandwidth), which is the total amount of work done in a given time. This is important to data center managers.

To maximize performance, we need to minimize the execution time.

$$\text{performance}_X = \frac{1}{\text{execution time}_X}$$

If X is n times faster than Y, then

$$\frac{\text{performance}_X}{\text{performance}_Y} = \frac{\text{execution time}_Y}{\text{execution time}_X} = n$$

Notice that decreasing response time almost always improves throughput.

11.2 Performance Factors

There are several performance factors that need to be considered. The CPU execution time (CPU time) is the time that the CPU spends working on a task. It doesn't include the time spent waiting for I/O or running other programs.

$$\text{CPU execution time} = \#\text{ CPU clock cycles} \times \text{clock cycle time} = \frac{\#\text{ CPU clock cycles}}{\text{clock rate}}$$

Then, we can improve performance by reducing the length of the clock cycle or the number of clock cycles required for a program.

> **Remark.** Clock rate is the inverse of clock cycle time:
>
> $$\text{Clock Cycle time} = \frac{1}{\text{Clock Rate}}$$

However, note that not all instructions take the same amount of time to execute. One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction.

$$\text{CPU clock cycles} = \#\text{ instructions} \times \text{clock cycles per instruction}$$

The average number of clock cycles each instruction takes to execute is the clock cycles per instruction. This is a way to compare two implementations of the same ISA.

Then, for the effective (average) CPI, we have

$$\text{CPI}_{\text{eff}} = \sum_{i=1}^{n} \text{CPI}_i \times \text{IC}_i$$

where IC_i is the percentage of the number of instructions of class i executed; CPI_i is the average number of clock cycles per instruction for that instruction class. n is the number of instruction classes.

Computing the overall effective CPI is done by considering the different types of instructions and their individual cycle counts, then averaging. The overall effective CPI varies by instruction mix, which is a measure of the dynamic frequency of instructions across one or many programs.

We also have a basic performance equation:

$$\begin{aligned}\text{CPU time} &= \text{Instruction count} \times \text{CPI} \times \text{clock cycle time} \\ &= \frac{\text{Instruction count} \times \text{CPI}}{\text{clock rate}}\end{aligned}$$

Here, the instruction count can be measured using profilers or simulators without knowing all of the implementation details. The CPI again varies by instruction type and ISA implementation, for which we must know the implementation details. The clock rate is usually given.

11.3 Workloads and Benchmarks

A set of programs that form a *workload* specifically chosen to measure performance is called a benchmark. The SPEC (System Performance Evaluation Cooperative) creates standard sets of benchmarks, starting with SPEC89.

To summarize the performance with a single number, we can use the SPEC ratio. They are averaged using the geometric mean (GM):

$$\text{GM} = n \cdot \sqrt{\sum_{i=1}^{n} \text{SPEC ratio}_i}$$

There are other performance metrics. For example, we can use power consumption, especially in the embedded market where battery life is important. However, for power-limited applications, the most important metric is energy efficiency.

Chapter 12

Memory

Now, we move on to another major component in the computer: Memory. The discussion of memory will include Cache, Main Memory, and Secondary Memory (Disk). In this chapter, we will look at the Main Memory and Disk.

12.1 Memory Hierarchy

Previously, we talked about combinational circuits and sequential circuits. The first always gives the same output for a given set of inputs, while the latter stores information and provides output depending on the stored information. Therefore, we need to have materials to store information.

The maximum size of memory is determined by the addressing scheme. For example, a 16-bit address can only address $2^{16} = 65,536$ memory locations. Most machines are byte-addressable, meaning each memory address location refers to a byte. Most machines retrieve or store data in words.

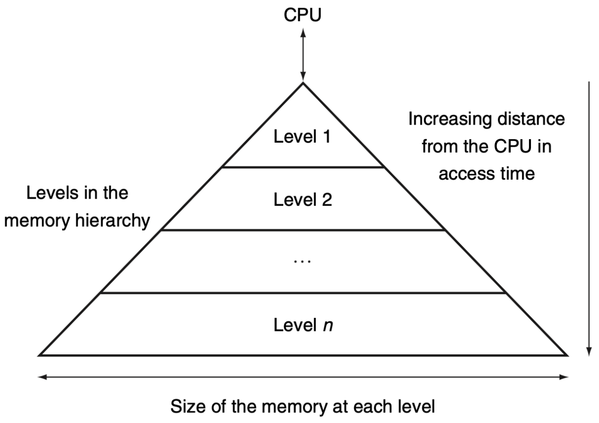
Also, there are some common abbreviations:

- $1\,\mathrm{k} \approx 2^{10}$ (kilo)
- $1\,\mathrm{M} \approx 2^{20}$ (Mega)
- $1\,\mathrm{G} \approx 2^{30}$ (Giga)
- $1\,\mathrm{T} \approx 2^{40}$ (Tera)

Data transfer takes place through the **MAR** (Memory Address Register) and the **MDR** (Memory Data Register). Both MAR and MDR are located in the processor and are connected to memory via a bus. By using the address from the MAR, we can locate the required memory location, and then read from or write to it using the MDR.

The processor usually runs much faster than main memory. Small memories are fast, while large memories are slow. Leveraging this characteristic, we use **cache memory** to store data in the processor that is likely to be accessed soon.

Main memory is limited in size. To address this, we use **virtual memory** to increase the apparent size of physical memory by moving unused sections to disk. The translation between virtual and physical addresses is handled by the **Memory Management Unit (MMU)**.


CPU
Level 1
Level 2
…
Level n
Levels in the memory hierarchy
Increasing distance from the CPU in access time
Size of the memory at each level

We can visualize the memory hierarchy, which follows an **inclusive property**, meaning that data in Level 1 cache is a subset of that in Level 2, and so on up the hierarchy.

We can view the memory hierarchy as follows:

- **Level 1 (L1)**: Cache using Static RAM (SRAM)
- **Level 2 (L2)**: Another level of cache, often using faster Dynamic RAM (DRAM)
- **Level 3 (L3)**: Main memory, typically DRAM
- **Level 4 (L4)**: Secondary memory such as flash storage or solid-state drives (SSD)

The first three levels consist of **volatile memory**, meaning they retain data only while power is on. In contrast, the secondary memory is **non-volatile** and retains data even when power is off.

As the distance from the CPU increases, so does the access time. Therefore, **Level 1 is the fastest**, and **Level 4 is the slowest**.

The above memory hierarchy works because of two types of locality. The first is **Temporal Locality** (locality in time), meaning that if a memory location is referenced, it will tend to be referenced again soon. It keeps the most recently accessed data items closer to the processor. The second is **Spatial Locality** (locality in space). If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon. Therefore, we move blocks consisting of contiguous words closer to the processor.

For example, a variable array follows **Spatial Locality**, while a variable value follows **Temporal Locality**.

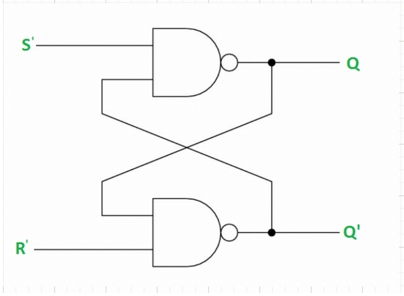
Before moving into the next section, we now introduce some terminologies.

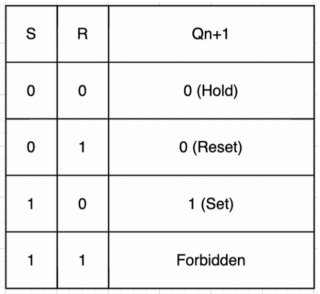
- **Random Access Memory (RAM)** has comparable access time for any memory location.
- **Block (or line)** is the minimum unit of information that is present in a cache.
- **Hit Rate** is the fraction of memory accesses found in a level of the memory hierarchy.
- **Miss Rate** is the fraction of memory accesses not found in a level of the memory hierarchy.
- **Hit Time** is the time to access the block plus the time to determine hit or miss.
- **Miss Penalty** is the time to replace a block in that level with the corresponding block from a lower level.
- **Bandwidth** is the amount of data transferred per second when transferring a block of data steadily.
- **Latency** is the amount of time to transfer the first word of a block after issuing the access signal.

> **Remark.** Note that Hit Time is much smaller than Miss Penalty.

12.2 Information Storage

We can store one bit of information using a pair of inverters, which is stable. However, if we want to change the value stored, we may replace the inverters with a NOR gate, resulting in the SR-Latch.





S	R	Qn+1
0	0	0 (Hold)
0	1	0 (Reset)
1	0	1 (Set)
1	1	Forbidden

> **Remark.** S is set, R is reset. Thus, if $S = 1$ and $R = 0$, the latch is set, and the output will be 1.

In SRAM cell, there are at least 6 transistors. This is used in most commercial chips, with a pair of weak cross-coupled inverters. Data is stored in the cross-coupled inverters, therefore, when power is applied, the data remains unchanged, making SRAM "static".

Since the data is stored in the cross-coupled inverters and doesn't need to be refreshed, it can be accessed quickly. However, SRAM cells require more transistors, which means they occupy more space on the chip.

In DRAM, there is only 1 transistor. It requires the presence of an external capacitor, and the modifications happen during the manufacturing process. For writing, we charge or discharge the capacitor. To read, charge redistribution takes place between the bit line and the storage capacitance.

The charge leaks away over time, so the data must be refreshed periodically to maintain it. This refresh cycle introduces additional latency compared to SRAM.

Because DRAM uses just a single transistor and a capacitor to store data, the DRAM cell is much smaller than an SRAM cell. This results in higher density, meaning more memory can be stored in the same physical space, making DRAM cheaper and suitable for larger memory sizes. So it's commonly used for main memory in computers.

12.3 Random Access Memory

In the SRAM cell array, all rows are connected to a row decoder, and all columns are connected to a column selector and I/O circuits. Each intersection of the row and column represents a 6-T SRAM cell. One memory row holds a block of data, so the column address selects the requested bit or word from that block.

Here we call the row the **word line**, which refers to the horizontal lines. The vertical column lines are called **bit-lines**, which carry the data.

The arrangement in DRAM is similar, but each intersection represents a 1-T DRAM cell. The column address selects the requested bit from the row in each plane, with the row address determining which row is accessed first. The data in each cell is stored as a charge in a capacitor, and a transistor is used to access the charge.

The more common type used today is **Synchronous DRAM** (SDRAM), which uses a clock to synchronize its operations. This synchronization allows the refresh operation to become transparent to the user. Additionally, all control signals needed for operation are generated internally within the chip.

Normal SDRAM operates once per clock cycle, transferring data on either the rising or falling edge of the clock. In contrast, Double Data Rate (DDR) SDRAM transfers data on both the rising and falling edges of the clock, effectively doubling the data transfer rate compared to standard SDRAM.

Static RAM (SRAM) retains its state as long as power is applied. It is fast, consumes low power, but is costly to manufacture, which results in smaller capacity. SRAM is typically used for **Level 1 (L1)** and **Level 2 (L2)** caches inside processors.

Dynamic RAM (DRAM) stores data as an electrical charge on a capacitor. Because the charge leaks away over time, DRAM must be periodically refreshed to maintain the stored data. In exchange for this need for refreshing, DRAM offers much higher density compared to SRAM.

Going back to the previous memory hierarchy, the aim is to produce fast, large, and cost-effective memory. Therefore, **Level 1 (L1)** and **Level 2 (L2)** caches are usually implemented with **SRAM**, while the main memory is typically **DRAM**. This design leverages the principle of locality of reference to enhance performance.

In summary, **DRAM** is slow but cheap and dense, while **SRAM** is fast but expensive and not very dense.

12.4 Interleaving

A memory controller is normally used to interface between the memory and the processor. **DRAMs** have a slightly more complex interface as they require refreshing, and they usually have time-multiplexed signals to reduce the number of pins. **SRAM** interfaces are simpler and may not require a memory controller.

The memory controller accepts a complete address and the read or write signal from the processor. Then, the controller will generate the **Row Access Strobe** (RAS) and **Column Access Strobe** (CAS) signals. The high-order address bits, which select a row in the cell array, are provided first under the control of the **RAS** signal. Then, the low-order address bits, which select a column, are provided on the same address pins under the control of the **CAS** signal.

Based on the address given, data lines are connected directly between the processor and the memory.

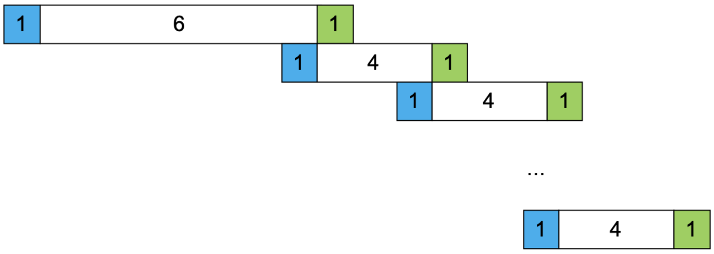
SDRAM needs refreshing, but the refresh overhead is only less than 1 percent of the total time available to access the memory.

Again, as mentioned before, the processor and cache are fast, but the main memory is slow. Thus, we try to hide access latency by interleaving memory accesses across several memory modules. Each memory module has its own Address Buffer Register and Data Buffer Register, so they operate somewhat independently.

To perform memory module interleaving, two or more compatible memory modules are used. Within a memory module, several chips are used in "parallel".

For example, suppose we have a cache read miss and need to load from main memory. Assume a cache with an 8-word block, i.e., the cache line size is 8 words. Then, assume it takes one clock to send the address to DRAM memory and one clock to send the data back. In addition, DRAM has a 6-cycle latency for the first word, with each of the subsequent words in the same row taking only 4 cycles. Then for a single memory read, we have

$$1 + 6 + 1 = 8 \text{ cycles.}$$

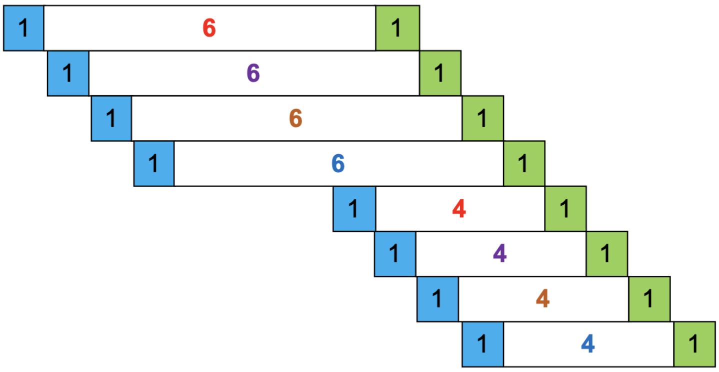


And for the case of non-interleaving, there is no overlapping in cache access. Then all subsequent words in DRAM need 4 cycles. For example, if there are 8 reads, the total time required will be:

$$1 + 1 \times 6 + 7 \times 4 + 1 = 36 \text{ cycles}$$

However, for four-module interleaving, we have

$$1 + 6 + 1 \times 8 = 15 \text{ cycles}$$



12.5 Secondary Memory

Magnetic disk is long-term, non-volatile storage. It is the lowest level of memory, which is slow but large and cheap. It has a rotating platter coated with a magnetic surface, with a movable read or write head to access the information. Its latency is the average seek time plus the rotational latency. Its bandwidth is the peak transfer rate of formatted data from the media.

We also have read-only memory (ROM). The memory content is fixed and cannot be changed easily, making it useful for bootstrapping a computer, since RAM is volatile when power is removed. We need to store a small program in such memory to start the process of loading the OS from a hard disk into the main memory.

Flash storage is also non-volatile and faster than disks. Flash devices have greater density, higher capacity, and lower cost per bit. It can be read and written. Flash cards are made from flash chips.

In summary, there are some common RAM types: SRAM, DRAM, SDRAM, and DDR SDRAM. We need to consider the principle of locality. Also, the memory hierarchy follows:

$$\text{Register} \rightarrow \text{Cache (SRAM)} \rightarrow \text{Main Memory (DRAM)} \rightarrow \text{Disk} \rightarrow \text{Tape.}$$

Chapter 13

Cache

We here continue the discussion on Memory.

13.1 Introduction

Cache-Main Memory mapping is a way to record which part of the Main Memory is currently in the cache. There are two design concerns: efficiency and effectiveness. We want to make the fast determination of cache hits or misses, and make full use of the cache, which increases the probability of cache hits.

Then, we must answer two questions: how do we know if a data item is in the cache? If it is, how do we find it?

Cache size is much smaller than the main memory size. Thus, a block in the main memory maps to a block in the cache, where both are constructed from many blocks. Since the number of blocks in the cache is smaller than in the main memory, this relationship is called a **many-to-one mapping**.

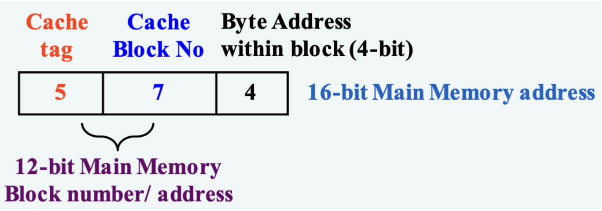
13.2 Direct Mapping

One way to perform the mapping is called **direct mapping**. Here, we consider a main memory address that is 16 bits wide. It is divided into three fields:

- **Cache tag**: occupies 5 bits (values from 0 to 31)
- **Cache block number**: occupies 7 bits (values from 0 to 127)
- **Byte address within the block**: occupies 4 bits

Thus, there are in total $2^4 = 16$ bytes in a block, $2^7 = 128$ Cache blocks, and $2^{(7+5)} = 4096$ Main Memory blocks.

The **Cache tag** is used to identify which **specific** block from main memory is currently stored in the cache. This is important because many main memory blocks can map to the same position in the cache. The **Cache Block Number** determines the location of the block in the cache and is used to index the cache. The last few bits of the address select the target word within the block.


Cache tag
Cache Block No
Byte Address within block (4-bit)
5
7
4
16-bit Main Memory address
12-bit Main Memory Block number/ address

For example, for a block j of Main Memory, it maps to Block ($j \mod 128$) of the Cache. A cache hit occurs if the tag matches the desired address.

One way to understand this is by using a **hash table** analogy. Many values can hash to the same index. If the desired value is found in the corresponding cache block, it is considered a **hit**; otherwise, it is a **miss**.

For a given address `t, b, w`, we can check if it is already in cache by comparing `t` with the tag in block `b`. If the tags do not match, it results in a cache miss. In that case, we replace the current block at `b` with a new one from Memory block `t, b`.

For example, if the CPU is looking for `A7B4`, where MAR = `1010011110110100`, it goes to cache block `1111011` and checks if the tag is `10100`. If so, it's a cache hit. Otherwise, the block is fetched into cache row `1111011`.

In direct mapping, a block in main memory is always mapped to the same cache location. If there is a conflict in address, at most one of them can be mapped to the cache. If a cache block is not mapped, its `valid` bit is set to 0.

The number of bits includes both the storage for data and for the tags. For a direct-mapped cache with 2^n blocks, n bits are used for the index. For a block size of 2^m words (2^{m+2} bytes), m bits are used to address the word within the block.

2 bits are used to address the byte within the word.

We can also calculate the size of the tag field as follows (for the case that it uses a 32-bit address):

$$\text{Tag size} = 32 - (n + m + 2)$$

The total number of bits in a direct-mapped cache is:

$$2^n \times (\text{block size} + \text{tag field size} + \text{valid field size})$$

The **Cache tag** is used to identify which **specific** block from main memory is currently stored in the cache. This is important because many main memory blocks can map to the same position in the cache. The **Cache Block Number** determines the location of the block in the cache and is used to index the cache. The last few bits of the address select the target word within the block.

Then, for the total number of bits in a direct-mapped cache, we have

$$2^{10} \times (4 \times 32 + (32 - 10 - 2 - 2) + 1) = 147\,\text{Kbits}$$

> **Remark.** 1 word = 4 bytes = 32 bits

13.3 Associative Mapping

A Main Memory block can also be in an arbitrary Cache block location. For example, we can divide the 16-bit main memory address into two parts, with the tag occupying 12 bits, and the byte address occupying 4 bits. Then, all 128 tag entries must be compared with the address tag in parallel.

For example, if the CPU is looking for `A7B4`, where MAR = `1010011110110100`, it will check if the tag `101001111011` matches one of the 128 cache tags. If yes, then it's a cache hit. Otherwise, we will load the block into the BINGO cache row.

We can combine the Associative and direct mapping. We again divide the field into 3 parts, where the first 6 bits are the tag, the following 6 bits are the set number, and the last 4 bits are the byte address. We can derive the set number by using $j \mod 64$, where a cache with k-blocks per set is called a k-way set-associative cache.

Again, if the CPU is looking for `A7B4`, where MAR = `1010011110110100`, it goes to check the set `111011`. It will see if one of the two tags in the set `111011` is `101001`. If yes, then it is a cache hit. Otherwise, we will load the block into the BINGO cache row.

For a fixed-size cache, the tag is used to perform tag comparison, the index is used to select the set, and the block offset is used to select the word within the block. For direct mapping, we have smaller tags, but for fully associative mapping, the tag consists of all the bits except the block and byte offsets. This is distinguished by a "freedom line" between the tag field and the index field.

We call it fully associative (with maximum freedom) when each block can be mapped to any location, and to find the block, we need to compare the tag each time we perform a lookup.

13.4 Replacement

When finding a block in the cache, if there is a read hit, that is desirable. However, when there is a read miss, we need to stall the pipeline, fetch the block from the next level in the memory hierarchy, install it in the cache, send the requested word to the processor, and then let the pipeline resume.

Here, we consider only the case when cache write hits. There are two cases:

The first one is **write-through**, where the cache and memory must remain consistent, and we always write the data into both the cache block and the next level in the memory hierarchy. We can use a write-buffer and stall only when the buffer is full to speed up the process. Notice that memory updates can be handled by the buffer, which operates independently of the processor pipeline. This method is easier to implement. Notice that read misses don't result in writes.

Another case is **write-back**. In this method, we write the data only into the cache block, and update the memory hierarchy only when that cache block is *evicted* (replaced). Thus, we need a **dirty bit** for each data cache block.

> **Remark.** A **dirty bit** is a flag used in cache memory management to track whether the data in a cache block has been modified (written to) since it was last loaded from main memory.

Here we return to the mapping techniques discussed in the previous sections. For **direct mapping**, the position of each block is fixed. Whenever replacement is needed (i.e., a cache miss leading to a new block being loaded), the choice is obvious, and thus no replacement algorithm is needed.

However, for **associative** and **set-associative** mapping, we need to decide which block to replace, aiming to keep the ones likely to be used again in the near future.

One strategy we can use is the **Least Recently Used (LRU)** policy. For example, for a 4-block cache, we use a $\log_2 4 = 2$-bit counter for each block. We reset the counter to 0 whenever the block is accessed, and the counters of other blocks in the same set are incremented. On a cache miss, we replace (or uncache) a block whose counter reaches 3.

Another strategy we can use is **random replacement**. We choose a random block to replace, which is easier to implement at high speed.

13.5 More on Cache

In this section, we take a look on three examples.

> **Example** (Example 1 - Analysis). Consider the following code.

```
short A[10][4];
int sum = 0;
int j, i;
double mean;

for (j = 0; j <= 9; j++)  // forward loop
  sum += A[j][0];
mean = sum / 10.0;
for (i = 9; i >= 0; i--)  // backward loop
  A[i][0] = A[i][0]/mean;
```

Here we consider only the data, with the cache having space for 8 blocks. `A[10][4]` is an array of words located at addresses `7A00 - 7A27` in row-major order.

With direct mapping, after the first loop loads data into the cache, we will only have two hits since all data in the first loop maps to block 0 and block 4, leading to constant replacements in the cache.

We can use associative mapping, where we apply the LRU (Least Recently Used) replacement rule. In this case, all the blocks in the cache will be utilized, and we will have cache hits for $i = 9, 8, \ldots, 2$. However, if the second loop is also a forward loop, there will be no hits.

Another mapping technique we can use is set-associative mapping. However, all the accessed blocks have even addresses, leading to a situation where only half of the cache blocks will be used. We again use the LRU rule for replacement, and only $i = 9, 8, 7, 6$ will result in cache hits. Again, if it is a forward loop, there will be no hits. Thus, in this case, random replacement would yield better average performance.

In this example, associative mapping performs the best. However, in practice, it is rare for such low hit rates to occur. Typically, set-associative mapping with an LRU replacement scheme is used.

Example (Total Bits Calculation). How many total bits are required for a direct-mapped cache with 16 KiB of data and 4-word blocks, assuming a 32-bit address? What about an associative-mapped cache?

1. Direct Mapping

For a direct-mapped cache with 16 KiB of data and 4-word blocks, we have $\frac{16}{4\times 4} = 1$ K blocks (1024 blocks).

Using the formula as before, we have

$$\begin{aligned}\text{Total number of bits} &= 2^n \times (\text{block size} + \text{tag field size} + \text{valid field size})\\ &= 2^{10} \times (4 \times 4 \times 8 + (32 - 10 - 2 - 2) + 1)\\ &= 2^{10} \times 147 \text{bits}\end{aligned}$$

Thus, the total number of bits in the cache is about $\frac{147}{32\times 4} \approx 1.15$ times as many as needed just for the storage of the data.

2. Associative Mapping

Again, for a cache with 16 KiB of data and 4-word blocks, we have $\frac{16}{4\times 4} = 1$ K blocks (1024 blocks). Using the formula as before, we have

$$\begin{aligned}\text{Total number of bits} &= 2^n \times (\text{block size} + \text{tag field size} + \text{valid field size})\\ &= 2^{10} \times (4 \times 4 \times 8 + (32 - 2 - 2) \text{ (without block ID)} + 1)\\ &= 2^{10} \times 157 \text{bits}\end{aligned}$$

Thus, the total number of bits in the cache is about $\frac{157}{32\times 4} \approx 1.27$ times as many as needed just for the storage of the data.

Example. We have designed a 64-bit address direct-mapped cache, and the bits of the address used to access the cache are shown below:

Tag	Index	Offset
63-10	9-5	4-0

1. What is the block size of the cache in words?

Solution: Since the offset is 5 bits, the block size $= 2^5 = 32$ bytes.

Thus, the block size in words is $32/8 = 4$ words.

2. Find the ratio between total bits required for such a cache design implementation over the data

storage bits.

Solution: Since the index is 5 bits, we have $2^5 = 32$blocks.

Thus, the cache stores $32 \times 4 \times 8 \times 8 = 8192$ bits.

Since each line contains 54 bits of tag and 1 bit of valid bit, the total bits required is $8192 + 54 \times 32 + 1 \times 32 = 9952$ bits.

3. Beginning from power-on, the following byte-addressed cache references are recorded as shown below. Find the hit ratio.

Hex	00	04	10	84	E8	A0	400	1E	8C	C1C	B4	884
Dec	0	4	16	132	232	160	1024	30	140	3100	180	2180

Solution:

Byte Address	Binary Address	Tag	Index	Offset	Hit / Miss
`0X00`	`0000 0000 0000`	`00`	`00000`	`00000`	M
`0X04`	`0000 0000 0100`	`00`	`00000`	`00100`	H
`0X10`	`0000 0001 0000`	`00`	`00000`	`10000`	H
`0X84`	`0000 1000 0100`	`00`	`00100`	`00100`	M
`0XE8`	`0000 1110 1000`	`00`	`00111`	`01000`	M
`0XA0`	`0000 1010 0000`	`00`	`00101`	`00000`	M
`0X400`	`0100 0000 0000`	`01`	`00000`	`00000`	M
`0X1E`	`0000 0001 1110`	`00`	`00000`	`11110`	M
`0X8C`	`0000 1000 1100`	`00`	`00100`	`01100`	H
`0XC1C`	`1100 0001 1100`	`11`	`00000`	`11100`	M
`0XB4`	`0000 1011 0100`	`00`	`00101`	`10100`	H
`0X884`	`1000 1000 0100`	`10`	`00100`	`00100`	M

Thus, the hit ratio is $\frac{4}{12} \times 100\% = 33\%$.

13.6 Performance

We summarize a bit about where a block is placed in the upper level and how it can be found.

Scheme name	Number of sets	Blocks per set
Direct mapping	# of blocks	1
Set associative	$\frac{\#\text{ of blocks}}{\text{Associativity}}$	Associativity
Fully associative	1	# of blocks

Scheme name	Location method	# of comparisons
Direct mapping	Index	1
Set associative	Index the set; compare its tag	Degree of associativity
Fully associative	Compare all tags	# of blocks

When a cache miss happens, for direct mapping, we have only one choice. However, for set-associative or fully associative caches, the choice is either random or follows the LRU rule. For higher levels of associativity, however, LRU is too costly.

Now, we consider the performance for different settings. Performance tells us how fast machine instructions can be brought into the processor and how fast they can be executed. Two key factors are performance and cost.

However, for a hierarchical memory system with cache, the processor is able to access instructions and data more quickly when the data needed are in the cache. Therefore, the impact of a cache on performance depends on the hit and miss rates.

A high hit rate of over 0.9 is essential for high-performance computers. A penalty is incurred because extra time is needed to bring a block of data from a slower unit to a faster one in the hierarchy. During this time, the processor is stalled. The waiting time depends on the details of the cache operation.

> **Remark.** Miss Penalty refers to the total access time experienced by the processor when a miss occurs.

For example, consider a computer where the access times to the cache and the main memory are t and $10t$, respectively. When a cache miss occurs, a block of 8 words is transferred from the main memory to the cache. It takes $10t$ to transfer the first word of the block, and the remaining 7 words are transferred at a rate of one word per t seconds.

The miss penalty then becomes:

$$t + 10t + 7t + t = 19t$$

Here, the first t is the initial cache access that results in a miss, $10t$ is the time to transfer the first word from main memory, $7t$ accounts for the transfer of the remaining 7 words, and the final t is the time to move the required word from the cache to the processor.

We can use one formula to compute the average memory access time:

$$\text{Average Memory Access Time} = h \times C + (1 - h) \times M$$

where h is the hit rate, C is the cache access time, and M is the miss penalty.

> **Example.** Assume we need 8 cycles to read a single memory word, and 15 cycles to load an 8-word block from main memory. The cache access time is 1 cycle. For every 100 instructions, statistically, 30 instructions are data read or write. For instruction fetch, we assume a 0.95 hit rate for 100 memory accesses. For the 30 memory accesses for data read or write, we assume a hit rate of 0.9.
>
> 1. What are the execution cycles without cache?
>
> **Solution:**
>
> $$\text{Execution cycle} = (100 + 30) \times 8 = 1040$$
>
> 2. What are the execution cycles with cache?
>
> **Solution:**
>
> $$\text{Execution cycle} = 100 \times [0.95 \times 1 + 0.05 \times (1 + 15 + 1)] + 30 \times [0.9 \times 1 + 0.1 \times (1 + 15 + 1)] = 258$$

In high-performance processors, two levels of caches are normally used: L1 and L2. L1 must be very fast, as it determines the memory access time seen by the processor. L2 cache can be slower, but it should be much larger than the L1 cache to ensure a high hit rate. Its speed is less critical because it only affects the miss penalty of the L1 cache.

For the average access time in such a system, we have:

$$h_1 \times C_1 + (1 - h_1) \times [h_2 \times C_2 + (1 - h_2) \times M]$$

Here, h_1 and h_2 are the hit rates, C_1 is the cache access time, C_2 is the miss penalty to transfer data from L2 cache to L1, and M is the miss penalty to transfer data from main memory to L2 and then to L1.

Before we talk about the two types of locality, we can first consider spatial locality. If all items in a larger block are needed in a computation, it is better to load these items into the cache in a single miss. However, larger blocks are effective only up to a certain size, beyond which too many items will remain unused before the block is replaced. Larger blocks also take longer to transfer and thus increase the miss penalty. Therefore, block sizes of 16 to 128 bytes are most popular.

The miss rate increases if the block size becomes a significant fraction of the cache size, because the number of blocks that can be held in the same cache is smaller, leading to more capacity misses.

Chapter 14

Virtual Memory

14.1 Overview

First, we introduce some terminology. A running program is called a process or a thread. The operating system (OS) controls the processes.

Physical memory may not be as large as the "possible address space" spanned by a processor. For example, a processor can address 4 GB with a 32-bit address. However, the installed main memory may only be 1 GB. How can we run many programs simultaneously when their total memory consumption exceeds the installed main memory capacity?

We can use main memory as a "cache" for the secondary memory. Each program is then compiled into its own virtual address space. This approach relies on the principle of locality.

In virtual memory, a virtual address is translated to a physical address during runtime. It enables efficient and safe sharing of memory among multiple programs, the ability to run programs larger than the size of physical memory, and code relocation, meaning that code can be loaded anywhere in main memory.

To share physical memory, a program's address space is divided into pages (fixed size) or segments (variable sizes). The frequently used blocks are copied into the cache.

In Virtual Memory, part of the process(es) are stored temporarily on the hard disk and brought into main memory as needed. This is done automatically by the operating system; the application program does not need to be aware of the existence of virtual memory (VM). The memory management unit (MMU) translates virtual addresses to physical addresses.

14.2 Virtual Memory

In address translation, memory is divided into pages of size ranging from 2 KB to 16 KB. If the page is too small, too much time is spent getting pages from disk. If the page is too large, a large portion of the page may not be used, but it will occupy valuable space in the main memory. This is similar to the issue we face when dealing with cache block size.

For hard disk, it takes a considerable amount of time to locate data on the disk. But once located, the data can be transferred at a rate of several MB per second.

An area in the main memory that can hold one page is called a page frame. The processor generates virtual addresses. The MS (high-order) bits are the virtual page number, and the LS (low-order) bits are the offset.

Information about where each page is stored is maintained in a data structure in the main memory called the **page table**. The starting address of the page table is stored in a page table base register. The address in physical memory is obtained by indexing the virtual page number from the page table base register.

By a combination of hardware and software, we can translate a virtual address into a physical address.

For each memory request, the first step is address translation. As mentioned before, a virtual address consists of a **virtual page number (VPN)** and a **page offset**. The virtual page number is translated into a **physical page number (PPN)**, while the offset remains unchanged.

To perform this translation, we use the **page table**, which stores the mapping between virtual and physical pages. The page table is typically stored in **main memory**. However, the use of a page table to translate addresses introduces one extra memory access. Thus, recent translations may be cached in the **Translation Lookaside Buffer (TLB)** for faster access.

This is a small cache that keeps track of recently used address mappings. Since it avoids accessing the slower main memory, it speeds up the address translation process. However, on a TLB miss, the system still needs to access the page table in main memory to perform the translation. The number of memory accesses ranges from 0 (if the translation is already cached in the TLB and the data is also in cache) to 2 (one for the page table lookup and one for the actual memory access).

In the TLB, there is a dirty bit, which indicates whether the page has been written to, and a reference bit, which indicates whether a page has been accessed.

If the required page is not in main memory (a **page fault** occurs), the operating system loads the required page from **secondary storage (disk)** into RAM and updates the page table accordingly.

Moreover, in the TLB, the organization can be fully associative, set-associative, or direct-mapped. The access time is faster than that of the cache due to its smaller size.

There are a few combinations:

TLB	Page Table	Cache	Possibility
Hit	Hit	Hit	Best case
Hit	Hit	Miss	Page table won't be checked
Miss	Hit	Hit	TLB miss, handled by Page table
Miss	Hit	Miss	TLB miss, handled by Page table, while data not found
Miss	Miss	Miss	Page fault
Hit	Miss	Miss / Hit	TLB translation is not possible if page is not in memory
Miss	Miss	Hit	Data is not allowed in cache if page is not in memory

> **Remark.** We do not access cache using virtual addresses because two programs may share data with different virtual addresses but the same physical address.

In the virtual memory address, the high-order bits are used to access the TLB, while the low-order bits are used as the index into the cache. The set number is in the page offset. Therefore, before we perform the translation using the page table, we can locate one set in the cache. Cache access and address translation can be carried out at the same time.

Notice that the TLB, which caches recent translations, is managed by hardware. Its access time is part of the cache hit time, and it may require an additional stage in the pipeline. The page table stores fault detection and updates, which are handled by both hardware and software: the dirty bit and reference bit are maintained by hardware, while page faults result in interrupts that are handled by software. Finally, disk placement is managed by software.

Chapter 15

Instruction Level Parallelism

15.1 Introduction

Here we again introduce some terminology.

We talked about pipeline before, which can be used to improve performance. To further improve performance, we use **superpipelining**, which increases the depth of the pipeline (breaking each task into smaller pieces) to raise the clock rate. However, adding more stages to the pipeline requires more forwarding or hazard-handling hardware and introduces greater pipeline latch overhead—i.e., the pipeline latch accounts for a larger and larger percentage of the clock cycle time.

Therefore, instead of just one instruction entering the pipeline per cycle, we allow multiple instructions to enter and be processed in parallel. If we fetch and execute more than one instruction at a time by expanding every pipeline stage to accommodate multiple instructions, this is called **Multiple-Issue**.

Since the instruction execution rate, i.e., cycles per instruction (CPI), can be less than 1 in this case, we instead use **instructions per cycle (IPC)**. For example, a 3GHz four-way multiple-issue processor can execute at a peak rate of 12 billion instructions per second, with a best-case CPI of 0.24 or a best-case IPC of 4.

Thus, we have **instruction-level parallelism (ILP)**, which is a measure of the average number of instructions in a program that a processor might be able to execute at the same time. This is mostly determined by the number of true data dependencies and procedural control dependencies in relation to the number of other instructions.

We also have **machine parallelism**, which is a measure of the ability of the processor to take advantage of the ILP of the program. This is determined by the number of instructions that can be fetched and executed at the same time.

In short, ILP is about what the program allows, while machine parallelism is about what the machine can do. To achieve high performance, we need both ILP and Machine Parallelism.

There are two different architectural approaches to exploiting instruction-level parallelism. One is called the **static multiple-issue processor** (VLIW), where compiler decides which instructions can run in parallel. Another style is the **dynamic multiple-issue processor** (superscalar), where the hardware (CPU) decides at runtime which instructions can run in parallel.

The static style has a faster runtime, but its performance is limited. For the dynamic style, there is a hardware penalty, and it requires complete knowledge of the program.

15.2 Dependencies

In chapter 9 and 10, we discuss three types of hazards. Structural hazards are caused by resource conflicts. A superscalar or VLIW processor has a much larger number of potential resource conflicts, where functional units may have to arbitrate for result buses and register-file write ports. However, resource conflicts can be eliminated by duplicating the resource or by pipelining the resource.

The second hazard is data hazards, which are caused by storage dependencies. The limitation is more severe in a superscalar or VLIW processor due to the low ILP.

The last hazard is the control hazard, which is caused by procedural dependencies. This is similar to data hazards, but even more severe. Thus, we need to use dynamic branch prediction to help resolve the ILP issue, which requires the combination of hardware and software.

We first take a look at data hazards.

For data hazards, there are three types. The first one is called **True Dependency (Read After Write - RAW)**. This happens when a later instruction uses a value that is not yet produced by an earlier instruction. The second one is called **Anti-dependency (Write After Read - WAR)**. This happens because the later instruction produces a data value that overwrites a data value used as a source in an earlier instruction that will be executed later. The last one is called **Output Dependency (Write After Write - WAW)**, where two instructions write to the same register or memory location.

> **Remark.**
>
> `R3 := R3 * R5` (RAW: Read After Write);
>
> `R4 := R3 + 1` (WAR: Write After Read);
>
> `R3 := R5 + 1` (WAW: Write After Write).

For example, in the following instruction sequence:

```
ADD R1, R2, R1
LW  R2, 0(R1)
LW  R1, 4(R1)
OR  R3, R1, R2
```

There are a few data dependency issues.

	RAW	WAR	WAW
R1	I1 -> I2 ; I1 -> I3; I3 -> I4	I1 -> I3 ; I2 -> I3	I1 -> I3
R2	I2 -> I4	I1 -> I2	N/A
R3	N/A	N/A	N/A

True dependencies represent the flow of data and information through a program. Antidependencies and output dependencies arise because of the limited number of registers, since programmers reuse registers for different computations, leading to storage conflicts.

Storage conflicts can be reduced by increasing or duplicating the troublesome resources. We can also provide additional registers (in more powerful CPUs) that are used to re-establish the correspondence between registers and values. Alternatively, registers can be allocated dynamically by the hardware in superscalar processors.

Register renaming is also a strategy. The processor renames the original register identifier in the instruction to a new register, which is not part of the visible register set. The hardware that performs renaming assigns a “replacement” register from a pool of free registers. It will later be released back to the pool when its value is superseded and there are no outstanding references to it.

For example, we have

```
R3 := R3 * R5               R3b := R3a * R5a
R4 := R3 + 1       ==>      R4a := R3b + 1
R3 := R5 + 1                R3c := R5a + 1
```

To resolve control dependencies, we use speculation, which allows the execution of future instructions that may depend on the speculated instruction. We can speculate on the outcome of a conditional branch (branch prediction) or speculate that a store (for which we don’t yet know the address) that precedes a load does not refer to the same address, allowing the load to be scheduled before the store (load speculation).

Thus, we must have hardware and software mechanisms for checking if the guess is correct, and recovering from the effects of the instructions that were executed speculatively if the guess was wrong.

We ignore exceptions caused by speculatively executed instructions until it is clear that they should actually occur.

15.3 VLIW

Static multiple-issue processors use the compiler to statically decide which instructions to issue and execute simultaneously. We have an *issue packet*, which is the set of instructions that are bundled together and issued in one clock cycle. It can be thought of as one large instruction with multiple operations (thus the name **Very Long Instruction Word**, or VLIW).

The mix of instructions in the packet is usually restricted, forming a single "instruction" with several predefined fields. The compiler performs static branch prediction and code scheduling to reduce or eliminate hazards.

VLIW processors have multiple functional units, multi-ported register files, and a wide program bus.

For example, consider a 2-issue RISC-V processor with a 2-instruction bundle. Each instruction bundle is now 64 bits, with the first half being an ALU operation or a branch, and the second half being a load or store instruction. Instructions are always fetched, decoded, and issued in pairs. If one instruction of the pair cannot be used, it is replaced with a `noop`.

Since there are two instructions issued per cycle, the processor requires four read ports, two write ports, and a separate memory address adder.

To expose ILP, we use (1) instruction scheduling, and (2) loop unrolling.

(1) Instruction scheduling

Consider the following loop code:

```
lp:  lw    $t0, 0($s1)      # $t0=array element
     addu  $t0, $t0, $s2    # add scalar in $s2
     sw    $t0, 0($s1)      # store result
     addi  $s1, $s1, -4     # decrement pointer
     bne   $s1, $0, lp      # branch if $s1 != 0
```

We must schedule the instructions to avoid pipeline stalls. Instructions in one bundle must be independent, and we must separate load-use instructions from their loads by one cycle. Notice that the first two instructions have a load-use dependency; the next two and last two have data dependencies. Here, we assume that branches are perfectly predicted by the hardware. Then we can reschedule the instructions as follows:

	ALU or branch	Data transfer	CC
`lp:`		`lw  $t0, 0($s1)`	1
	`addi  $s1, $s1, -4`		2
	`addu  $t0, $t0, $s2`		3
	`bne   $s1, $0, lp`	`sw  $t0, 0($s1)`	4
			5

Here, we can use only four clock cycles to execute five instructions. The CPI is $4/5 = 0.8$, and the IPC is $5/4 = 1.25$. Notice that `noop`s do not count towards performance.

(2) Loop Unrolling

We can also perform loop unrolling, where multiple copies of the loop body are made and instructions from different iterations are scheduled together as a way to increase ILP. After applying loop unrolling, we can schedule the resulting code. This helps to eliminate unnecessary loop overhead instructions, and scheduling helps to avoid load-use hazards. During unrolling, the compiler applies register renaming to eliminate all data dependencies that are not true data dependencies.

For example, consider the following loop:

```
lp: lw    $t0, 0($s1)     # $t0=array element
    lw    $t1, -4($s1)    # $t1=array element
    lw    $t2, -8($s1)    # $t2=array element
    lw    $t3, -12($s1)   # $t3=array element
    addu  $t0, $t0, $s2   # add scalar in $s2
    addu  $t1, $t1, $s2   # add scalar in $s2
    addu  $t2, $t2, $s2   # add scalar in $s2
    addu  $t3, $t3, $s2   # add scalar in $s2
    sw    $t0, 0($s1)     # store result
    sw    $t1, -4($s1)    # store result
    sw    $t2, -8($s1)    # store result
    sw    $t3, -12($s1)   # store result
    addi  $s1, $s1, -16   # decrement pointer
    bne   $s1, $0, lp     # branch if s1 != 0
```

After scheduling and unrolling, we have:

	ALU or branch	Data transfer	CC
`lp:`	`addi  $s1, $s1, -16`	`lw  $t0, 0($s1)`	1
		`lw  $t1, -4($s1)`	2
	`addu  $t0, $t0, $s2`	`lw  $t2, -8($s1)`	3
	`addu  $t1, $t1, $s2`	`lw  $t3, -12($s1)`	4
	`addu  $t2, $t2, $s2`	`sw  $t0, 0($s1)`	5
	`addu  $t3, $t3, $s2`	`sw  $t1, -4($s1)`	6
		`sw  $t2, -8($s1)`	7
	`bne  $s1, $0, lp`	`sw  $t3, -12($s1)`	8

Here, we use 8 clock cycles to execute 14 instructions. The CPI is $8/14 \approx 0.57$, and the IPC is $14/8 \approx 1.8$.

The compiler also supports VLIW processors. It packs groups of independent instructions into bundles, which is achieved through code reordering. The compiler uses loop unrolling to expose more ILP and applies register renaming to resolve name dependencies and avoid load-use hazards. While superscalar processors rely on dynamic prediction, VLIW processors primarily depend on the compiler for branch prediction. Loop unrolling reduces the number of conditional branches, and predication eliminates if-else branch structures by replacing them with predicated instructions. The compiler also predicts memory bank references to help minimize memory bank conflicts.

15.4 Superscalar

Dynamic multiple-issue processors use hardware at runtime to dynamically decide which instructions to issue and execute simultaneously.

We can have **Instruction Fetch and Issue**, where instructions are fetched, decoded, and issued to a functional unit to wait for execution. We define the **Instruction Lookahead Capability** as the ability to examine instructions beyond the current one. As soon as the source operands and the functional units are ready, the result can be calculated. We also define the **Processor Lookahead Capability** as the ability to complete execution of issued instructions beyond the current instruction, i.e., the processor can continue executing later instructions even if earlier ones are not yet finished, as long as it's safe to do so.

After an instruction is done executing, when it is safe to write back results to the register file or change the machine state, we perform **Instruction Commit**.

The **Instruction Fetch and Decode Units** are required to issue instructions in order so that dependencies can be tracked. The **Commit Unit** is responsible for writing results to registers and memory in the program fetch order.

If exceptions occur, only the registers updated by instructions before the one causing the exception will be modified. If branches are mispredicted, the instructions executed after the mispredicted branch do not change the machine state, since the commit unit ensures that incorrect speculation is corrected.

Although the frontend (fetch, decode, and issue) and backend (commit) of the pipeline run in order, the functional units are free to initiate execution whenever the required data is available. This is known as out-of-order execution. It allows instructions to be executed out of order, thereby increasing the amount of instruction-level parallelism (ILP).

With out-of-order execution, a later instruction may execute before a previous instruction, so the hardware needs to resolve both write-after-read (WAR) and write-after-write (WAW) data hazards.

```
lw    $t0, 0($s1)
addu  $t0, $t1, $s2
...
sub   $t2, $t0, $s2
```

For example, as shown in the code above, if the `lw` write to `$t0` occurs after the `addu` write, then the `sub` gets an incorrect value for `$t0`. The `addu` has an output dependency on the `lw`, which is a WAW (write-after-write) hazard. The issuing of the `addu` might have to be stalled if its result could later be overwritten by a previous instruction that takes longer to complete.

In summary, although ILP works, it is not as effective as we would like it to be. Some dependencies are hard to eliminate, and some parallelism is difficult to expose. However, speculation can help if done well.

To achieve high performance, we need both machine parallelism and instruction-level parallelism through techniques such as superpipelining, VLIW, and superscalar architectures. A processor's instruction issue and execution policies impact the available ILP, and register renaming can help resolve storage dependencies.