CENG3420 Computer Organization & Design

Ryan Chan

April 27, 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 $= \dfrac{1}{\left[1 + \left(\frac{\text{Defects per area} \times \text{Die area}}{2}\right)\right]^2}$

> **Remark.** Note that the defects on average = Defects per unit area × 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.

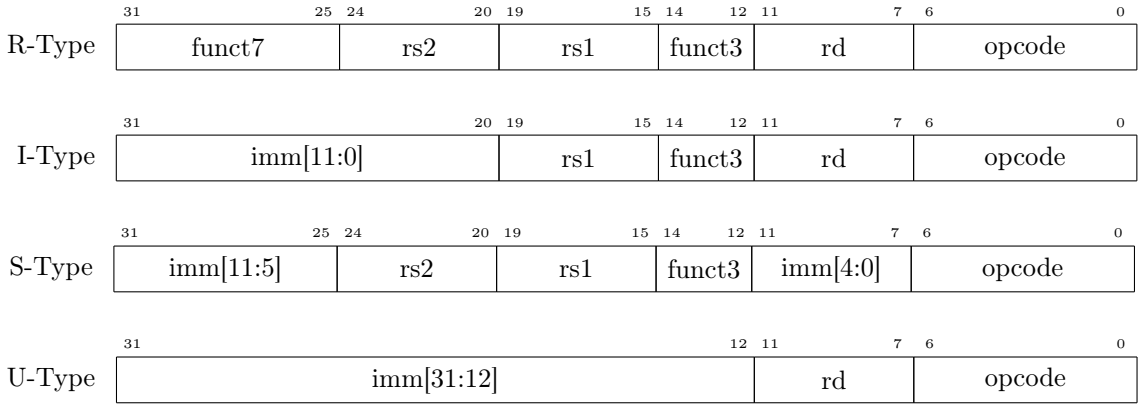
There are four design principles in RISC-V:

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



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:

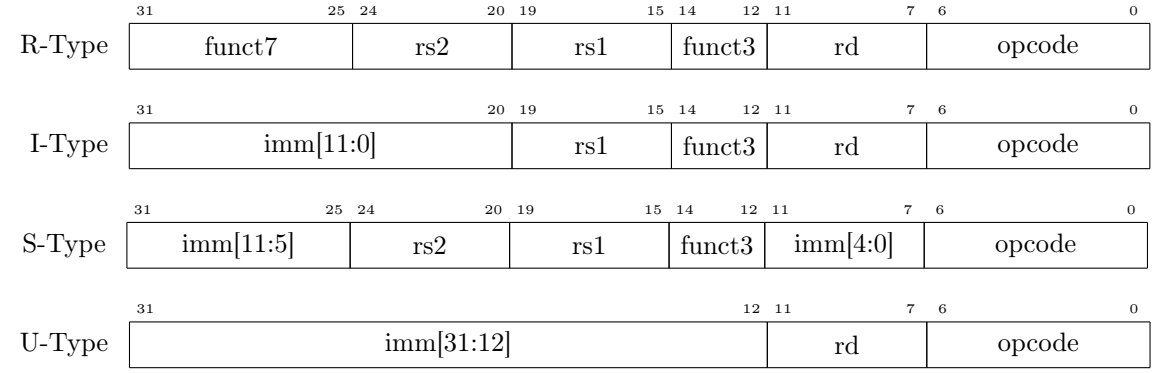
Return address (ra): Used to save the function return address, usually PC + 4.

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:



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.

```
1  .global _start
2
3  .text
4  _start:
5          li a1, 20
6          li a2, 23
7          add t0, a1, a2
8          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:                          Line 7: 10100 -> 1010000   # after slli 2 bits
5          li a1, 20                Line 8: 10111 -> 01011     # after srli 1 bits
6          li a2, 23
7          slli t0, a1, 2
8          srli t1, a1, 1
```

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.

```
1  .global _start
2
3  .data
4  a: .word 1 2 3 4 5
5
6  .text
7  _start:
8          la a1, a
9          lw t0, 0(a1)
10         lw t1, 4(a1)
11         lw t2, 8(a1)
12         lw t3, 12(a1)
13         lw t4, 16(a1)
14         addi t4, t4, 1
15         sw t4, 20(a1)
16         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:

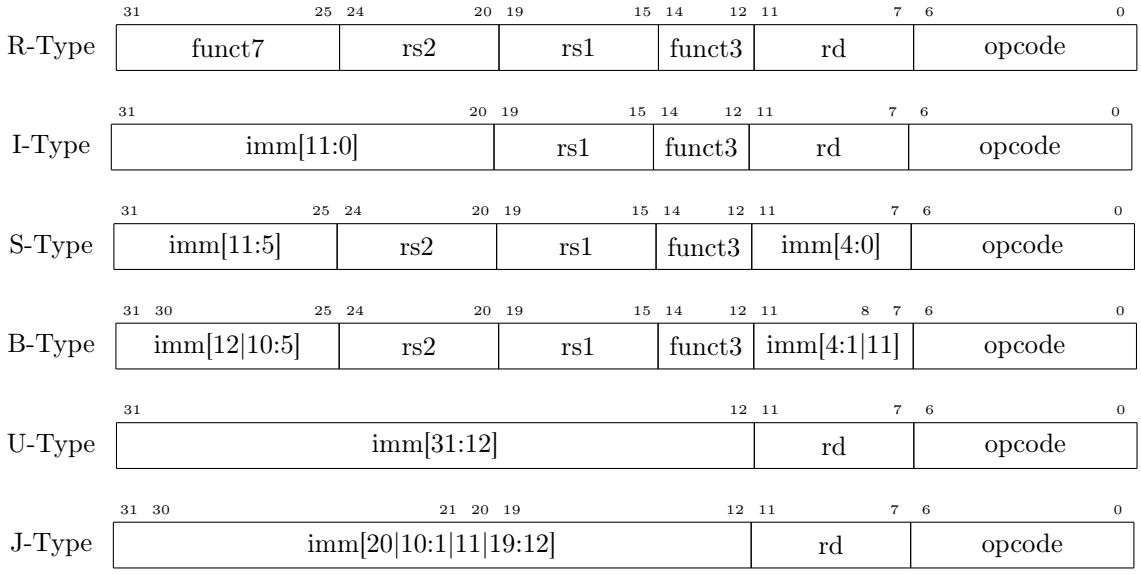
```
0xFFFF90FF    4
0x009012A0    0
```

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:

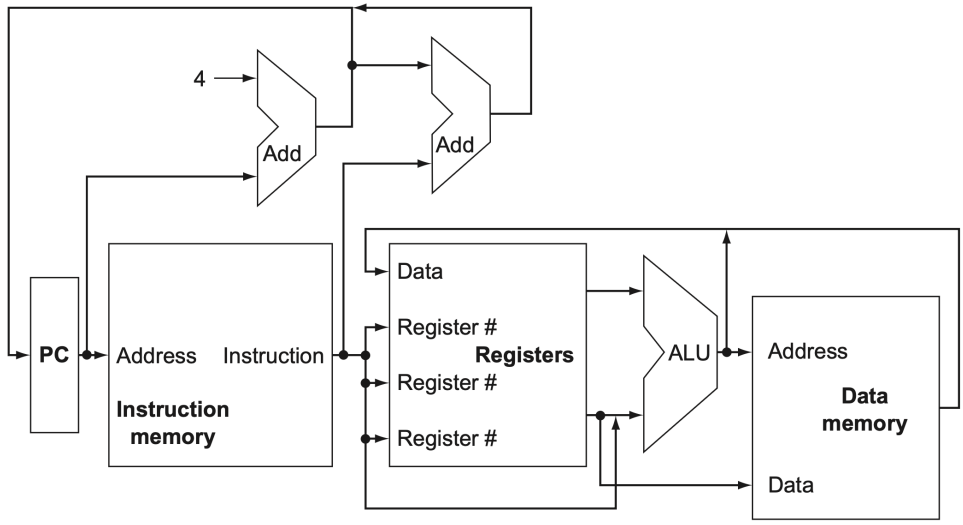
A	B	Output (A AND B)
0	0	0
0	1	0
1	0	0
1	1	1

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:

- $S = A \oplus B \oplus$ 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.

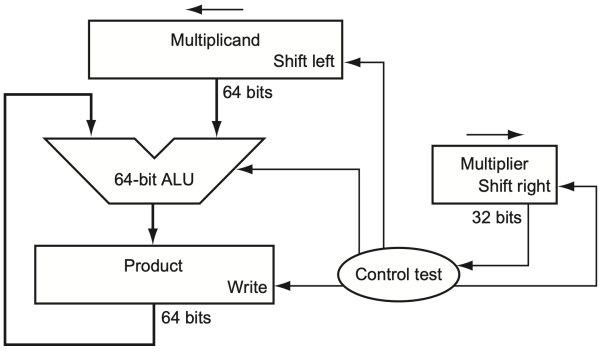
Overflow occurs when the result is too large to be represented. For example, adding two positive numbers yields a negative, adding two negative numbers gives a positive, subtracting a negative from a positive gives a negative, or subtracting a positive from a negative gives a positive. This leads to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

6.3 Multiplication and Division

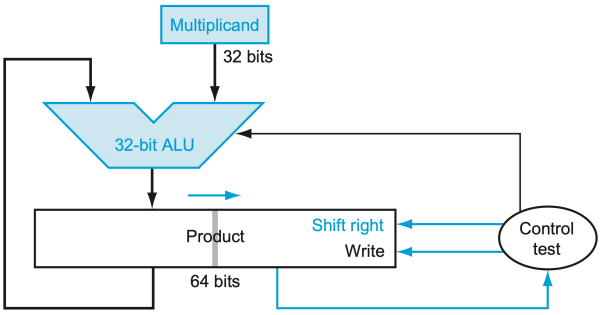
6.3.1 Multiplication

Multiplication is more complicated than addition. It can be accomplished by shifting and adding. For an n-bit $\times$ 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

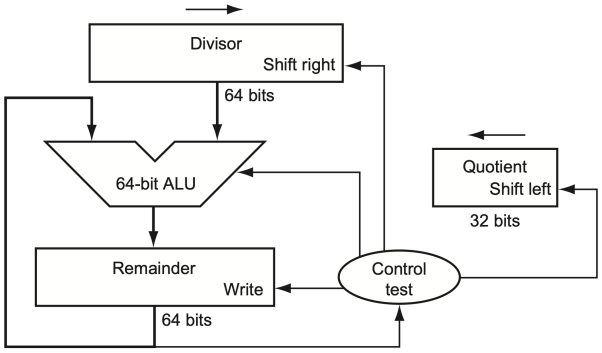
0010 × 0011				
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	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	0000	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	0000	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.

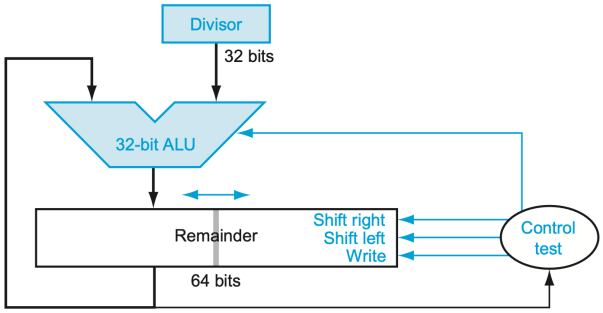
6.3.2 Division

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

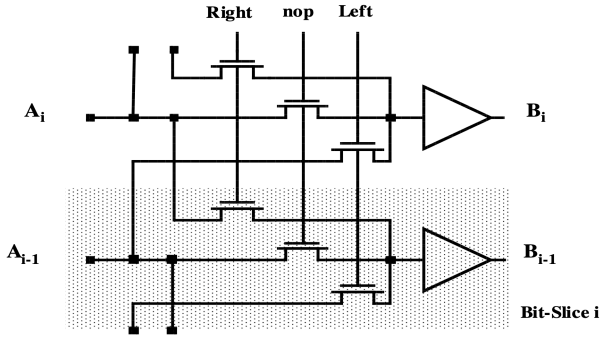
Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in rs1, and the shift amount is encoded in the lower 5 bits of the I-immediate field.

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.

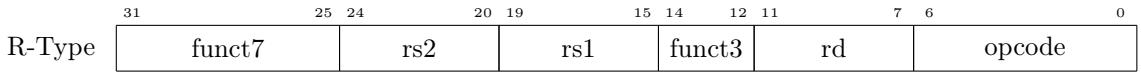
After the instructions are read, the processor decodes them. The fetched instruction's opcode and function field bits are sent to the control unit, which then generates control signals used in the future datapath. Next, two values are read from the Register File, with the register addresses contained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



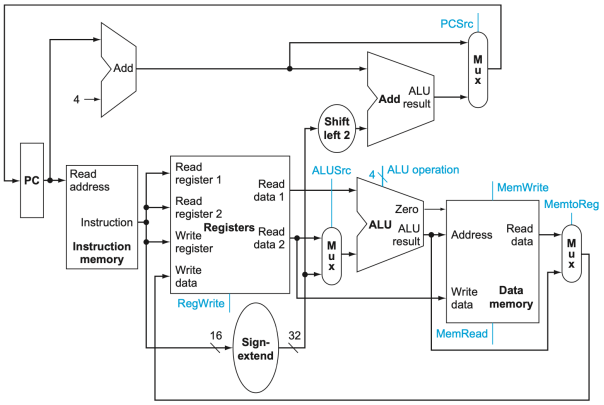
R-type instructions perform operations on values in `rs1` and `rs2`, then store the result back into the Register File. Note that the Register File is not written every cycle, so a write control signal is needed for the Register File.

8.2.2 I and S Format Operations

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.

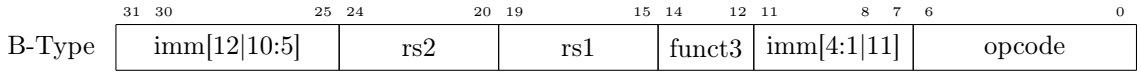
For store instructions, the value is read from the Register File during decode and written into the Data Memory. For load instructions, the value is read from the Data Memory and then stored into the Register File.

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



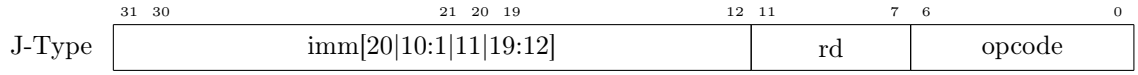
As shown above, after decoding, the sign is extended in the lower part. Above is the clock for the PC value, which executes the branch instruction. In the rightmost mux, it selects the source. It is activated only for `lw` instructions. Additionally, only for `sw`, the MemWrite signal will be 1 (High),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

8.2.3 B Format Operations



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



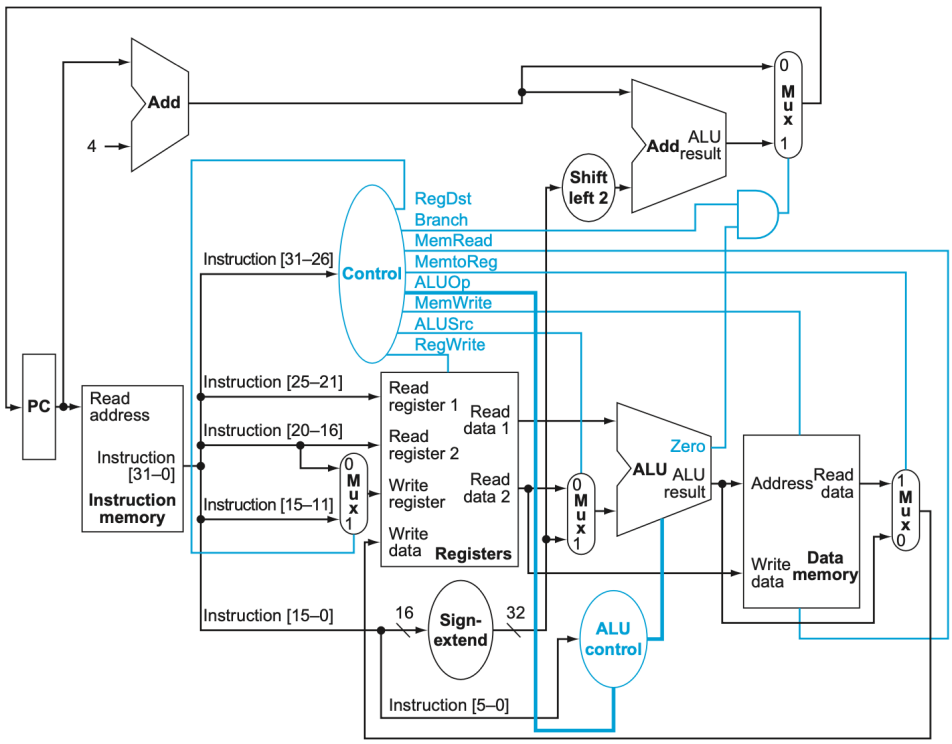
The J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Since `imm[0]` is always 0, we don't have it in the instruction.

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:

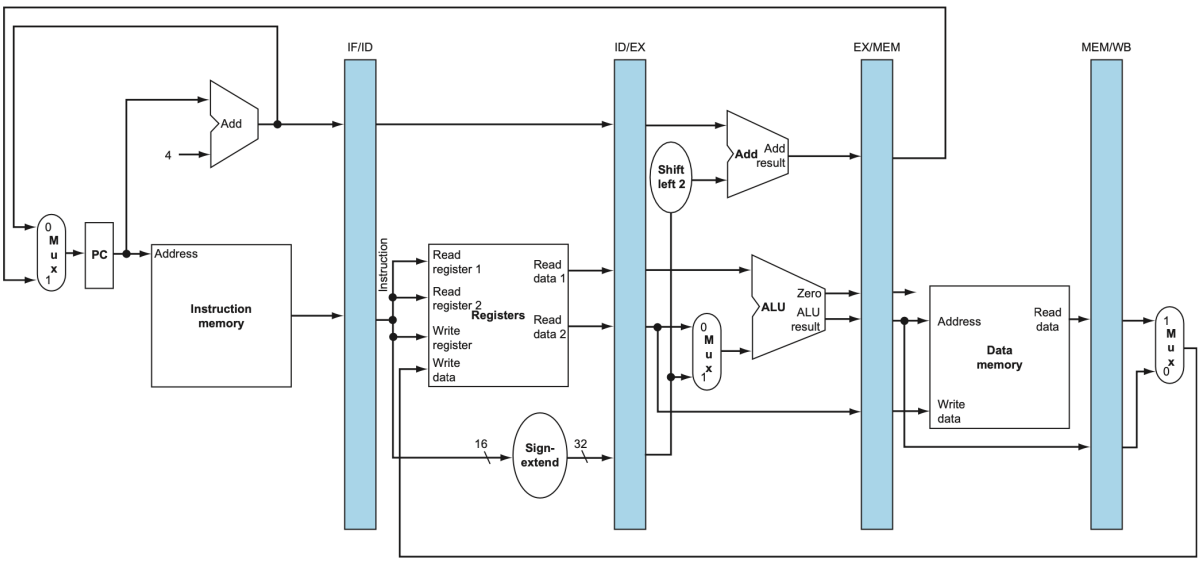
1. All instructions have the same length.

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

3. Memory operations occur only in loads and stores.

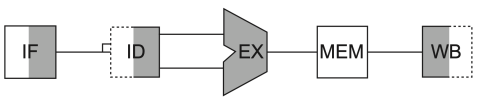
4. Each instruction writes at most one result, and it does so in the last few pipeline stages.

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.

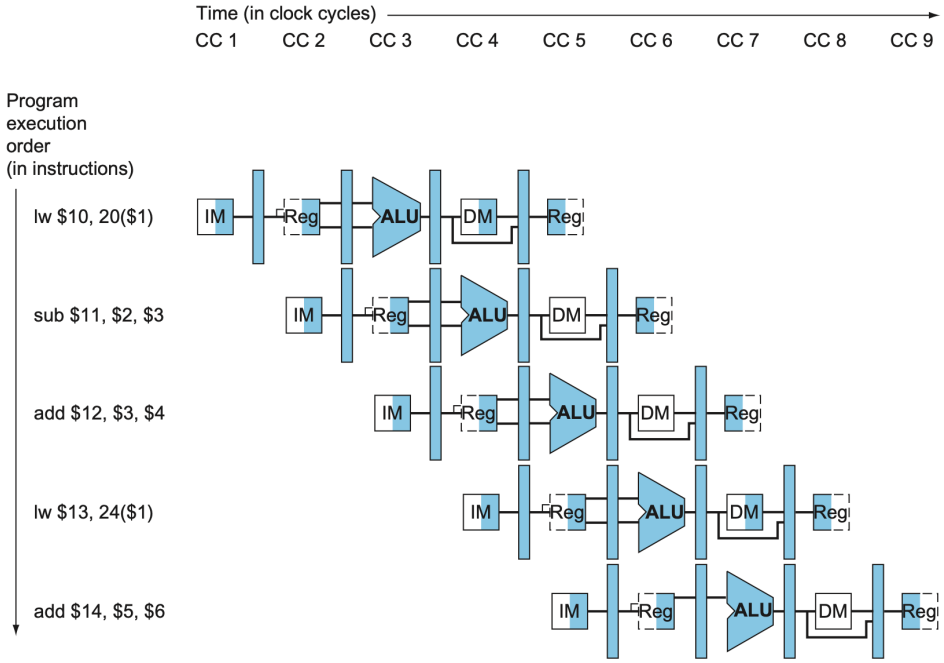
However, pipelines can cause issues, as they may introduce hazards. There are three possible pipeline hazards: structural hazards, caused by a busy resource; data hazards, where data is attempted to be used before it is ready; and control hazards, where control actions depend on the outcome of a previous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

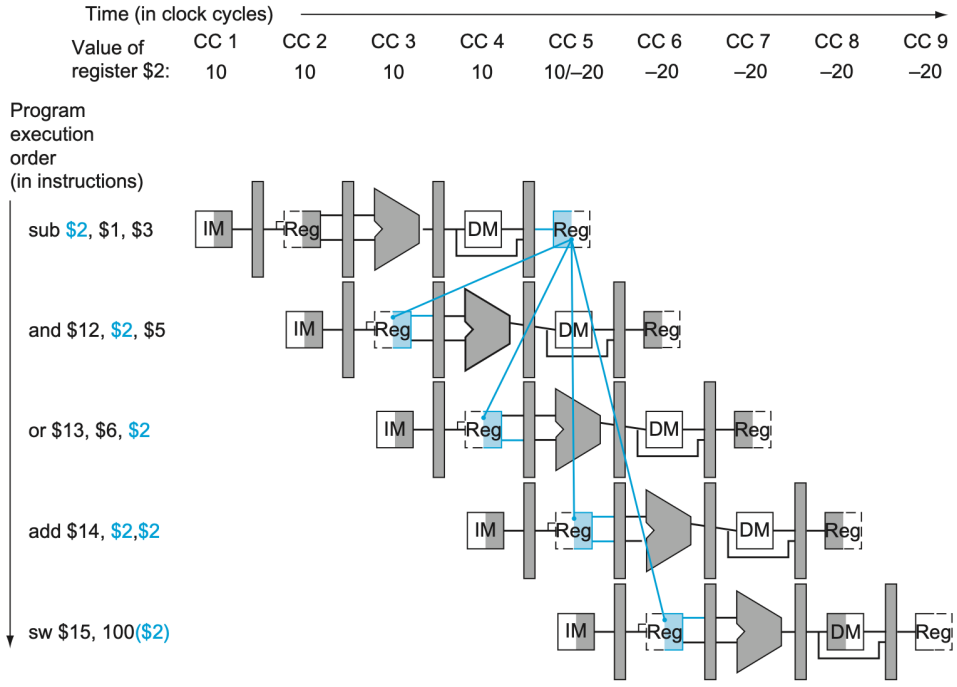
Here we continue the discussion on pipeline hazards.

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.

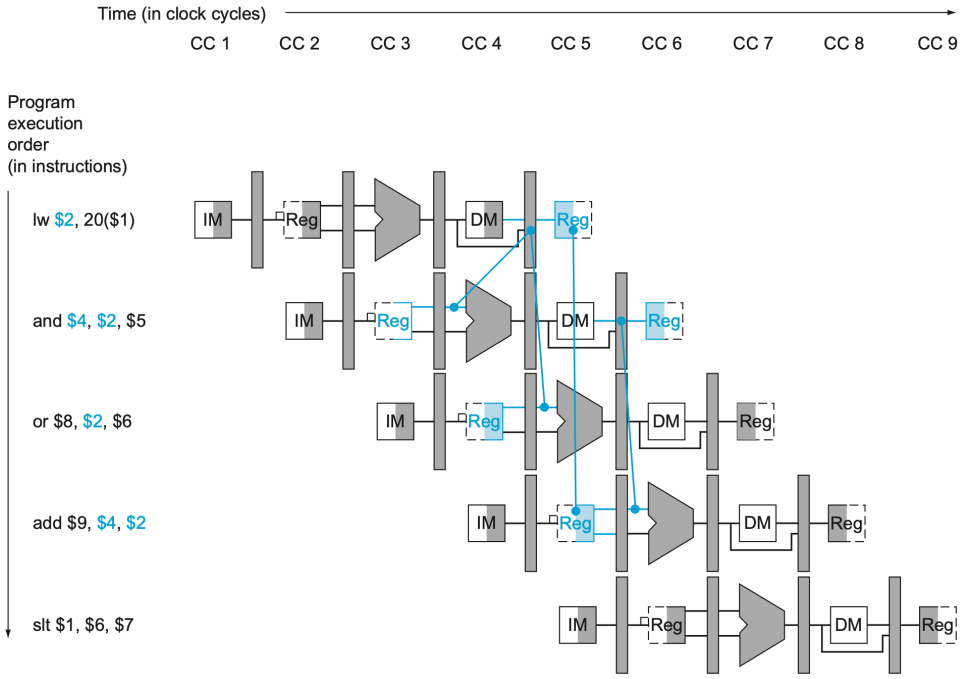
Data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline. There are two main types of data hazards:

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 `sub` instruction depends on the result of the `lw` instruction, leading to a data hazard.



To solve this problem, there are two main methods:

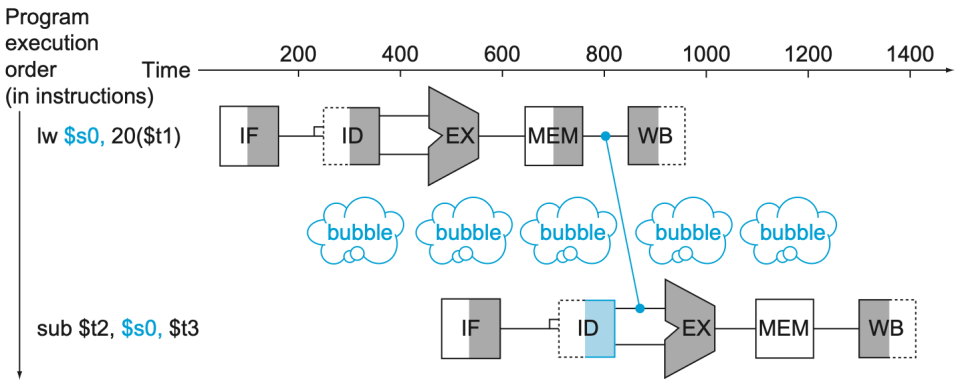
1. Insert NOP / Stall

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.

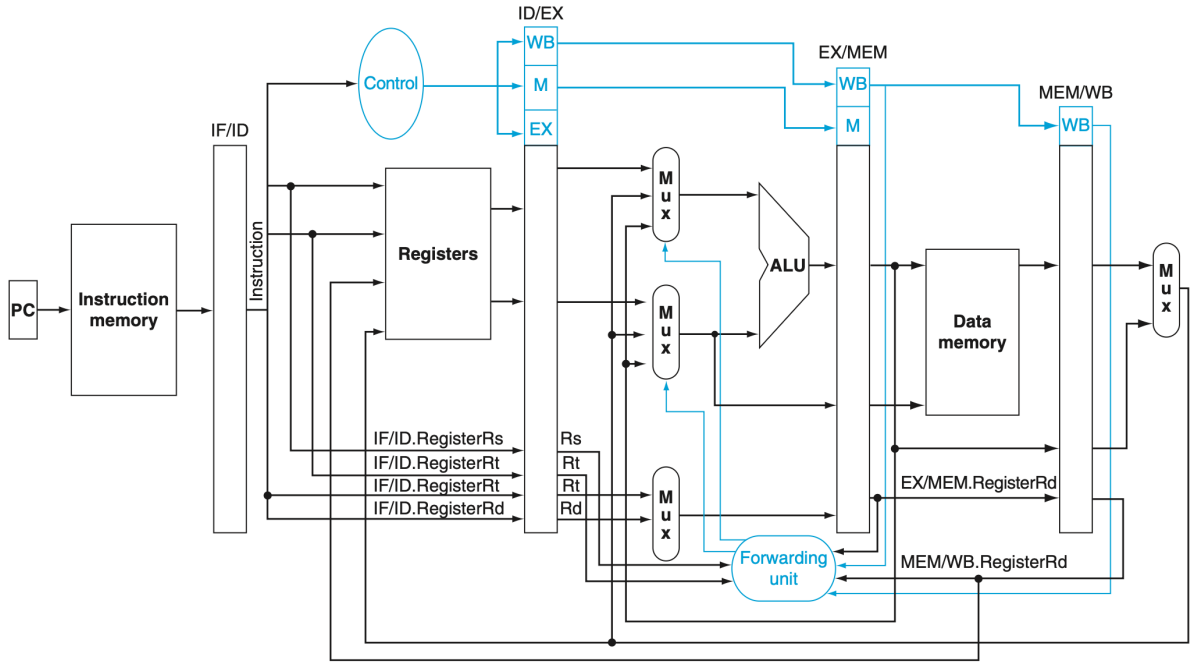
2. Forwarding

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.

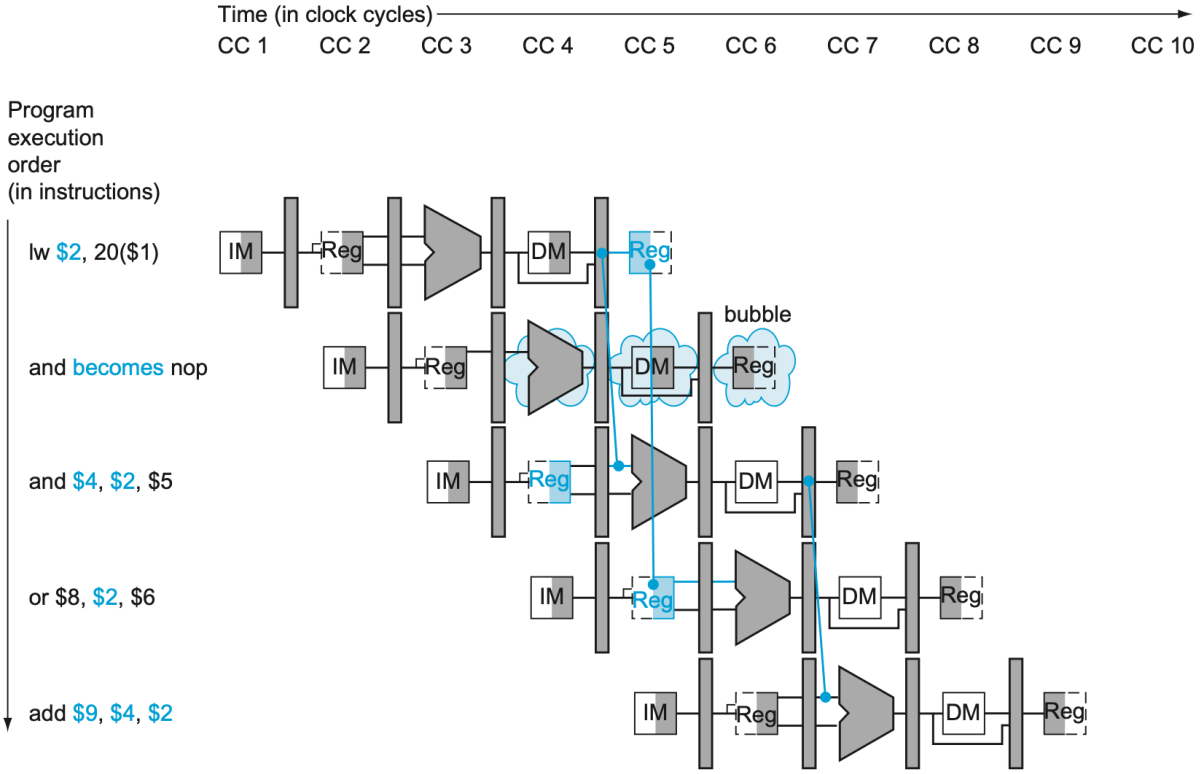
However, sometimes a combination of both methods is required. As shown below, after the `lw` instruction, we need to use both stalling and data forwarding to ensure that the data is transferred correctly.



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:



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

10.2.1 Pipeline Stalls

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.

However, control hazards occur less frequently than data hazards, and there is nothing as effective against control hazards as forwarding is against data hazards.

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.

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. This would easily fix the issue. However, as with data hazards, it 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** occurs when an instruction in the pipeline is replaced with a NOP instruction.

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 cycle. This will help reduce the delay of branch instructions.

To reduce the stalls, we can move the branch decision hardware back to the EX stage. This would reduce 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.

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

- The compiler moves an instruction that is not affected by the branch (a safe instruction) immediately after the branch, thereby hiding the branch delay.

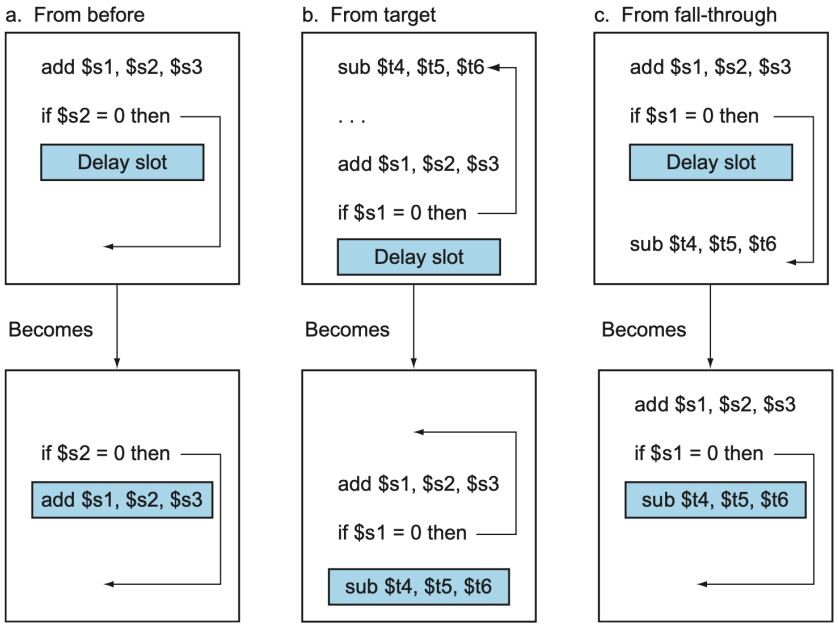
- With deeper pipelines, the branch delay grows, requiring more than one delay slot.

- Delayed branches have lost popularity compared to more expensive but more flexible (dynamic) hardware branch prediction.

- The growth in available transistors has made hardware branch prediction relatively cheaper.

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 the instruction, which reduces the IC (Instructions Count). In (b) and (c), the subtraction instruction may need to be duplicated, which increases the IC. However, they must be able to execute the subtraction when the branch fails.



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 to see the actual branch outcome.

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, 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" 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).

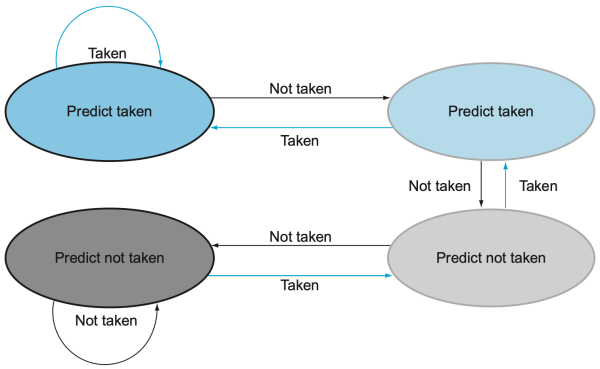
If the prediction is correct, stalls can be avoided no matter which direction they go.

We also have two types of predictor.

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

- Assume `predict bit` = 0 to start (indicating 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.

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

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

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.

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

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.

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

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:

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{clock cycle time}$$

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{clock rate}}$$

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.

11.2 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 also 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

Chapter 13

Cache

Chapter 14

Cache Disc

Chapter 15

Virtual Machine

Chapter 16

Instruction-Level Parallelism