CENG3420 Computer Organization & Design

Ryan Chan

March 7, 2025

Abstract

This is a note for **CENG3420 Computer Organization & Design** for self-revision purpose ONLY.
Some contents are taken from lecture notes and reference book.
Mistakes might be found. So please feel free to point out any mistakes.
Contents are adapted from the lecture notes of CENG3420, prepared by Bei Yu, as well as some online
resources.

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.
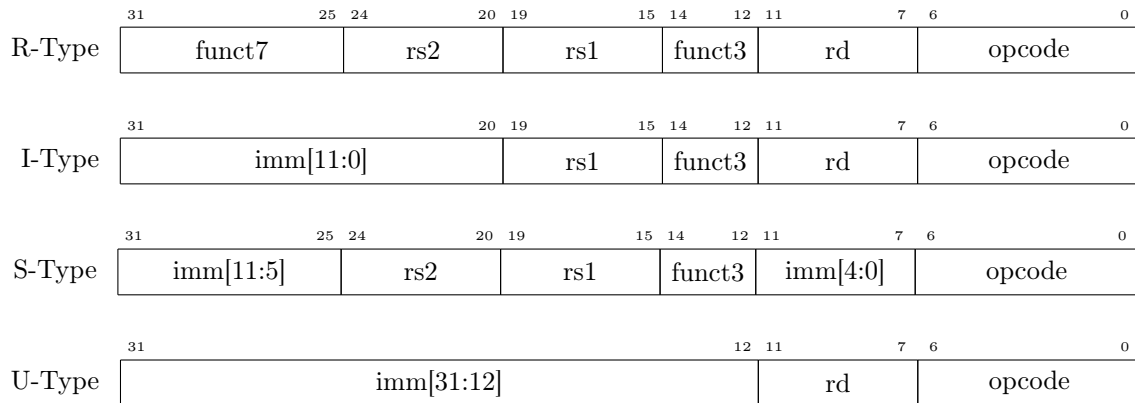
There are four design principles in RISC-V:

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

R-Type

funct7	rs2	rs1	funct3	rd	opcode

I-Type

imm[11:0]	rs1	funct3	rd	opcode

S-Type

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

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	

5

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:

```
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:

```
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
> ```
>
> The above 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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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.** Suppose that in memory, we have:
>
> ```
>     0x10000010    16
>     0x01000402    8
>     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:
>
> ```
>     0x10000010    16
>     0x01000402    8
>     0xFF12FFFF    4
>     0x009012A0    0
> ```
>
> 3. `t0 =  0x00000090`, New memory:
>
> ```
>     0x10000010    16
>     0x01000402    8
>     0xFFFF90FF    4
>     0x009012A0    0
> ```

Chapter 4

Control

Chapter 5

Logic basis

Chapter 6

Arithmetic Logic Unit

Chapter 7

Datapath

Chapter 8

Floating Number

Chapter 9

Pipeline

Chapter 10

More on Pipeline

Chapter 11

Performance

Chapter 12

Memory

Chapter 13

Cache

Chapter 14

Cache Disc

Chapter 15

Virtual Machine

Chapter 16

Instruction-Level Parallelism