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.
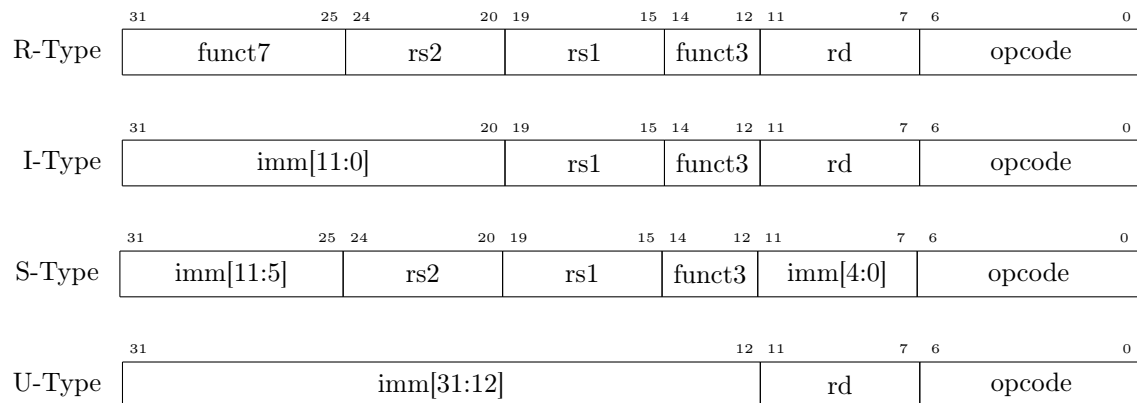
There are four design principles in RISC-V:

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

	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:

R-Type

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

I-Type

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

S-Type

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

U-Type

31 12	11 7	6 0
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:

```
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:                          This will give the result:
> 5          li a1, 20                 t0 = 0x2b, t1 = 0xfffffffd
> 6          li a2, 23
> 7          add t0, a1, a2
> 8          sub t1, a1, a2
> ```
>
> **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:                          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:

```
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                            a1 = 10100, a2 = 10111
> 4   _start:                          Line 7:   t0 = 10100 & 10111  ->  10100
> 5          li a1, 20                 Line 8:   t1 = 10100 | 10111  ->  10111
> 6          li a2, 23                 Line 9:   t2 = 10100 ^ 10111  ->  00011
> 7          and t0, a1, a2            Line 10:  t3 = 10100 & 10010  ->  10000
> 8          or t1, a1, a2             Line 11:  t4 = 10111  100001 ->  110111
> 9          xor t2, a1, a2
> 10         andi t3, a1, 0x12
> 11         ori t4, a2, 0x21
> ```

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:                         t0 = 0x01, t1 = 0x02
> 8           la a1, a                t2 = 0x03, t3 = 0x04
> 9           lw t0, 0(a1)            t4 = 0x06, t5 = 0x06
> 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)
> ```
> **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.** 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.

	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 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	

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

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

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:

```
  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                          Line 5:   a0 = 1
4   _start:                        Line 6:   a1 = 1
5         li a0, 1                 Line 7:   t0 = 20
6         li a1, 1                 Line 8:   t1 = 23
7         li t0, 20                Line 9:   t0 != t1 -> goto inst1
8         li t1, 23                Line 10 & 11 -> ignored
9         bne t0, t1, inst1        Line 12:  a0 = 3
10        addi a0, a0, 1           Line 13:  t0 != 0  -> goto end
11        beq t0, t1, inst2        Line 14 -> ignored
12        inst1:  addi a0, a0, 2   Line 15:  a0 = 2
13        bne t0, zero, end
14        inst2:  addi a0, a0, 3
15        end:    sub a0, a0, a1
```

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

```
  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                          Line 5:   a0 = 1
4   _start:                        Line 6:   t0 = 20
5         li a0, 1                 Line 7:   t1 = 23
6         li t0, 20                Line 8:   t0 < t1 -> a1 = 1
7         li t1, 23                Line 9:   a0 == a1 -> goto inst1
8         slt a1, t0, t1           Line 10:  ignored
9         beq a0, a1, inst1        Line 11:  a0 = 4
10        addi a0, a0, 2
11  inst1:  addi a0, a0, 3
```

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:

```
  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:

```
  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:

```
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
>
> $$\text{while (save[i] == k) i += 1;}$$
>
> 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.

```
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:                              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:  assign 8 bytes in stack
8         addi t1, a2, 0                           (from high to low)
9         j end                          Line 12:  save argument in stack 4(sp)
10  add_two_numbers:                     Line 13:  save argument in stack 0(sp)
11        addi sp, sp, -8                Line 14:  a2 = 43
12        sw a0, 4(sp)                   Line 15:  load argument from stack 4(sp)
13        sw a1, 0(sp)                   Line 16:  load argument from stack 0(sp)
14        add a2, a0, a1                 Line 17:  free stack
15        lw a0, 4(sp)                   Line 18:  jump to Line 8
16        lw a1, 0(sp)                   Line 8:   t1 = 43
17        addi sp, sp, 8                 Line 9:   jump to Line 16
18        jalr zero, 0(ra)              Line 16:  t1 = 44
19  end:
20        addi t1, t1, 1
```

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:

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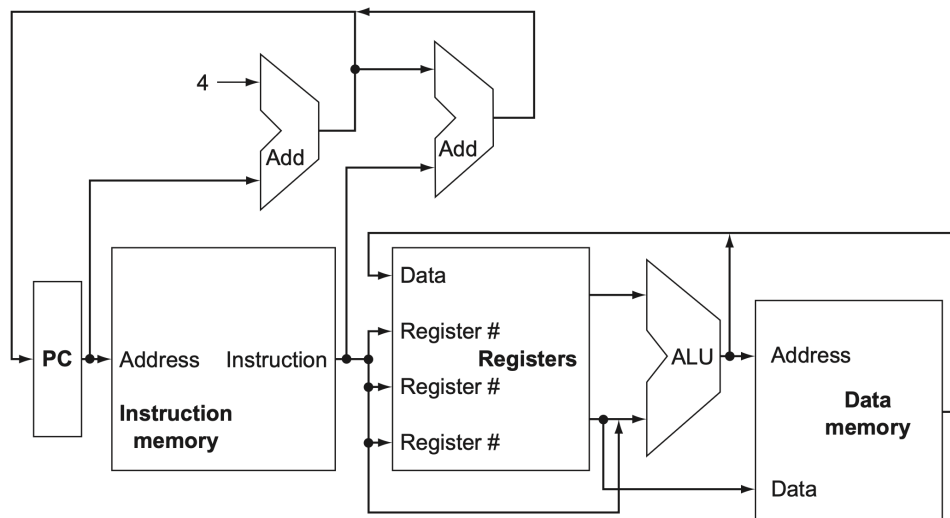
A	B	Output (A AND B)
0	0	0
0	1	0
1	0	0
1	1	1

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 $\times$ 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.
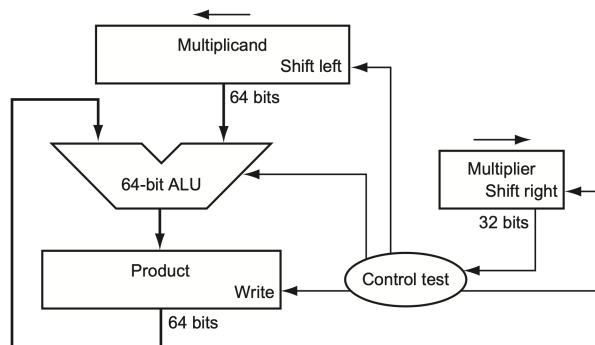
The refined version simplifies this by requiring only an n-bit adder for the same operation.
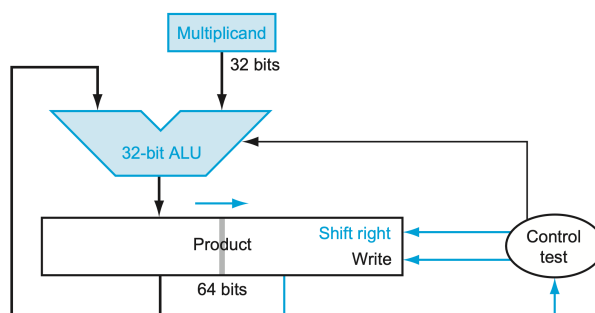
For example, when calculating $0010_2 \times 0011_2$, we have

0010 × 0011				
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001<u>1</u>	0000 0010	0000 0000
1	1a: 1 $\Rightarrow$ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	000<u>1</u>	0000 0100	0000 0010
2	1a: 1 $\Rightarrow$ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	000<u>0</u>	0000 1000	0000 0110
3	1: 0 $\Rightarrow$ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	000<u>0</u>	0001 0000	0000 0110
4	1: 0 $\Rightarrow$ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	000<u>0</u>	0010 0000	0000 0110

`mul` performs a 32-bit $\times$ 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.
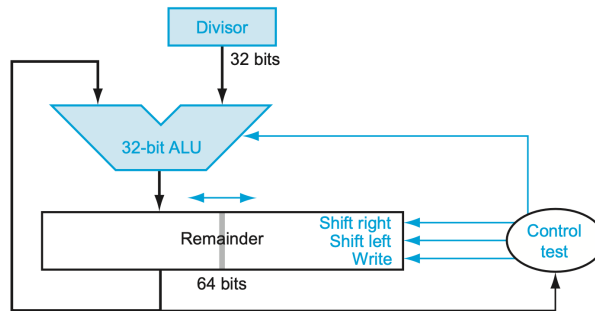
6.3.2 Division

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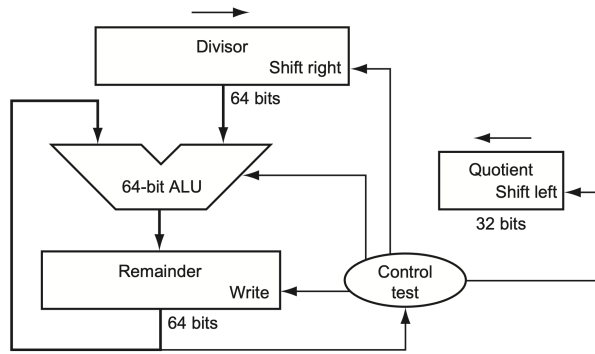
The refined version combines the Quotient register with the right half of the Remainder register.

`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.
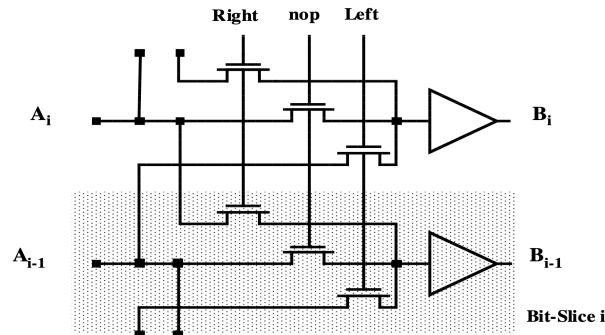
6.4   Shifter

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.

```
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

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