4.1

Date. No.

4.1

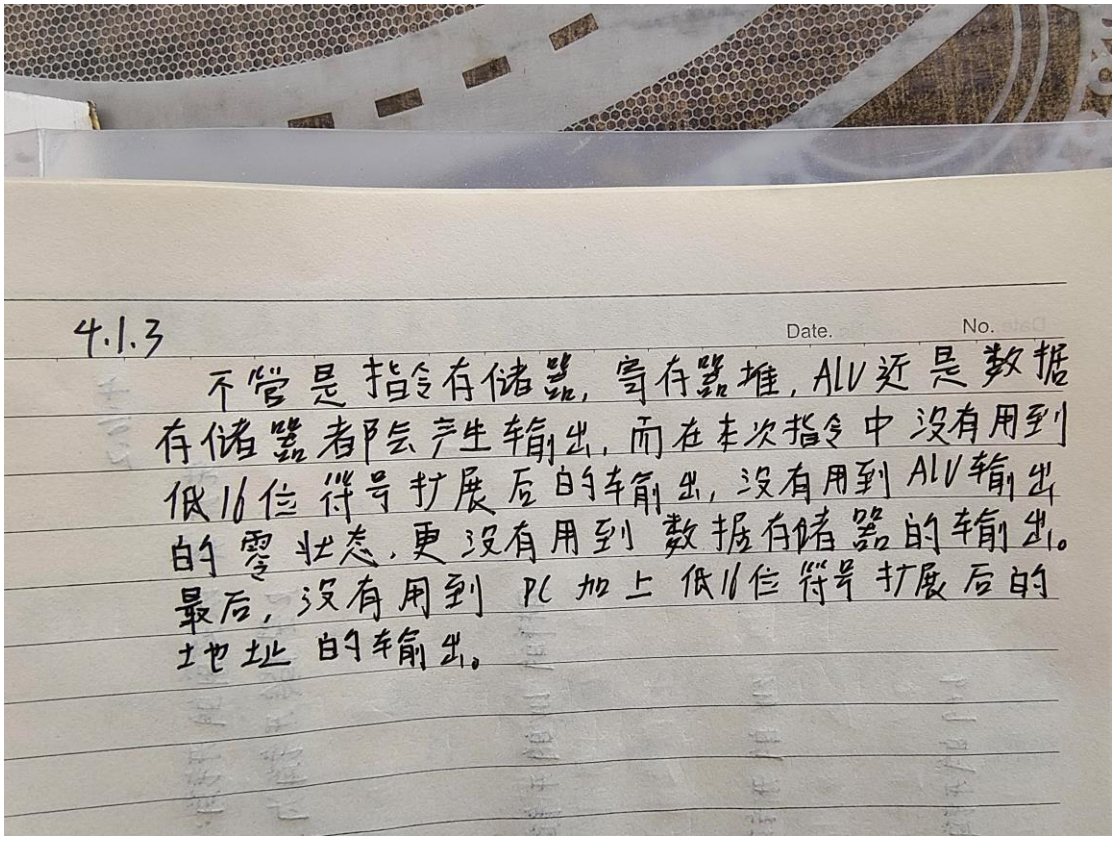
and rd, rs1, rs2.

4.1.1.

因为它写回的是 rd，所以 RegDst = 1.
因为没有分支，没有读存储器，没有写入存储器 所以 Branch，MemRead. MemWrite = 0
因为它写回寄存器的值来自 ALU. MemtoReg = 0
因为 ALU 的输入来自 rs 和 rt， ALUSrc = 0
因为最后要写回寄存器. RegWrite = 1
因为是 R 型指令，所以 ALUOp = 10.
因为是 and 指令，所以 ALU 控制单元的输出为 0000.

4.1.2.

首先取指令单元肯定会被用到，接着是指令译码单元和寄存器堆也肯定会被用到。然后需要在 ALU 里面计算与运算。因为这条指令并没有访问存储器，所以数据存储器并不会被用到。而是把 ALU 的输出写入 rd 寄存器，最后，因为不是分支指令或跳转指令，所以在加法模块里面计算 PC+4 就可以了。


4.1.3
Date.
No.
不管是指令存储器，寄存器堆，ALU还是数据存储器都会产生输出，而在本次指令中没有用到低16位符号扩展后的输出，没有用到ALU输出的零状态，更没有用到数据存储器的输出。最后，没有用到PC加上低16位符号扩展后的地址的输出。

4.3

4.3

R	I	lw	sw	BR	J
24%	28%	25%	10%	11%	2%

4.3.1

因为只有 lw 和 sw 会访问数据存储器，所以有 25%+10%=35% 的指令会用到

4.3.2

任何指令都会用指令存储器，所以是 100%

4.3.3

需要用符号扩展的是 I 型，lw，sw 和分支指令，所以是 28%+25%+10%+11%=74%.

4.3.4 如果是单周期数据通路的话，在需要的时候是处于空闲状态的，而如果在流水线数据通路

4.5

4.5.

指令：0x00c6ba23

二进制：0000 0000 1100 0110 1011 1010 0010 0011

op

4.5.1

因为op字段为000000，所以是R型指令，所以ALU控制单元的输入也就是ALUop为10

4.5.2

因为是R型，所以新地址为PC+4

4.5.3

判断图中哪个寄存器的多路选择器：

输入：Rt和rd的标号以及控制信号RegDst

00110

10111

1

输出：rd的标号：10111

判断ALU的输入是哪个的多路选择器

输入：readdata2和符号扩展后的结果以及控制信号ALUSrc

Rt寄存器数据

低16位符号扩展后的结果

0

输出：Rt寄存器数据

判断 返回的地址是 PC+4 还是 跳转地址的多路选择器

输入： PC+4 和 左移两位后的结果和 PC 的和 以及是否分支的控制信号

- PC+4
- PC+4+ 低16位 符号扩展 再左移两位
- 0

输出： PC+4

判断 要写回寄存器的是 ALU 的结果还是 数据寄存器的数据的多路选择器

输入： ALU运算的结果 和 数据寄存器的数据 以及控制信号 MemtoReg

- ALU 结果
- 数据寄存器
- 0

输出： ALU运算结果

因为是R型指令，所以两个寄存器的输出都要用到，所以寄存器的输出值都是各自寄存器里的值。

4.5.4

ALU的两个输入是 rs 和 rt 寄存器的数据

而 第一个加法器的输入则为 PC 和 4.

第二个 加法器并未用到。

4.5.5

寄存器堆从上到下的输入值为 rs 标号 rt 标号 rd 标号 以及要写回的值，也就是 00110 00110 10111 和 ALU运算的结果，而 PC 寄存器的输入为 PC+4

4.7

4.7.

4.7.1.

T(R) = PC + register setup + max{IM, Adder} + register file + register setup + MUX + control + MUX + ALU + max{single gate + MUX, MUX + register file + register setup}

= 30 + 20 + 250 + 150 + 20 + 25 + 50 + 25 + 200 + 25 + 150 + 20

= 965 ps

4.7.2

T(lw) = PC + register setup + max{IM, Adder} + control + MUX + register file + register setup + MUX + ALU + D-Mem + MUX + register file + register setup

= 30 + 20 + 250 + 50 + 25 + 150 + 20 + 25 + 200 + 250 + 25 + 30 + 20

= 1095 ps

4.7.3

T(sw) = PC + register setup + max{IM, adder} + control + register file + register setup + MUX + ALU + D-mem

= 20 + 30 + 250 + 50 + 150 + 20 + 25 + 200 + 250

= 995 ps

4.7.4

T(beq) = PC + register setup + max{IM, Adder} + control + max{register file + register setup, signex

+ max{shift left 2 + Adder, MUX + ALU} + single gate + MUX

= 20 + 30 + 250 + 50 + 150 + 20 + 25 + 200 + 5 + 25

= 775 ps

4.7.5

T(I) = PC + IM + control + Mux + register file + register setup + Mux + ALU +
max{signgate + Mux, Mux + register file + register setup}

= 20 + 30 + 250 + 50 + 25 + 150 + 20 + 25 + 200 + 25 + 150 + 20

= 965 ps

4.7.6

TC(CPU) = T(lw)

≈ 1095ps

4.10

R/I	lw	sw	beq
52%	25%	11%	12%

设 R和I 共有 52条，lw 有 25条，sw有11条，beq有12条。

4.10.1

先计算 改进前的时间：

因为 CPU 时钟周期要跟最长的 lw 一样，所以 = 1095ps

∴ TC(改进前) = 100 × 1095 = 109500ps

改进后 lw 和 sw 的数量变成了 25×0.88 和 11×0.88，也就是 22和 9.68

而寄存器堆的延迟变成了 160ps，也就是 CPU 时钟周期也增加了 10ps，变成了 1105ps

∴ T(改进后) = (52 + 22 + 9.68 + 12) × 1105

= 105726.4

所以加速比为 = $\frac{109500}{105726.4}$ = 1.04

4.10.2

先计算改进前的开销：

R型指令开销为：single register + control + register file + adder + Mux + Mux + ALU
+ sigle gate + Mux + Mux + register file + I-mem + single register
= 5 + 500 + 200 + 30 + 10 + 10 + 100 + 1 + 10 + 10 + 200 + 1000 + 5
= 2081

同样的 lw 的开销为：5 + 1000 + 30 + 500 + 10 + 200 + 100 + 10 + 100 + 1 + 10 + 5
+ 2000 + 10 + 200
= 4181

sw 的开销为：5 + 1000 + 30 + 500 + 200 + 100 + 10 + 100 + 1 + 10 + 5 + 2000
= 3961

beq 的开销为：5 + 1000 + 30 + 500 + 200 + 100 + 10 + 100 + 30 + 1 + 10 + 5
= 1891

I 型开销为 = 5 + 30 + 1000 + 500 + 10 + 200 + 100 + 10 + 100 + 1 + 10 + 5 + 10 + 200
= 2181

所以改进前的总开销为：

总开销 = 25 × 4181 + 11 × 3961 + 12 × 1891 + 52 × ((2081 + 2181) / 2)
= 281600

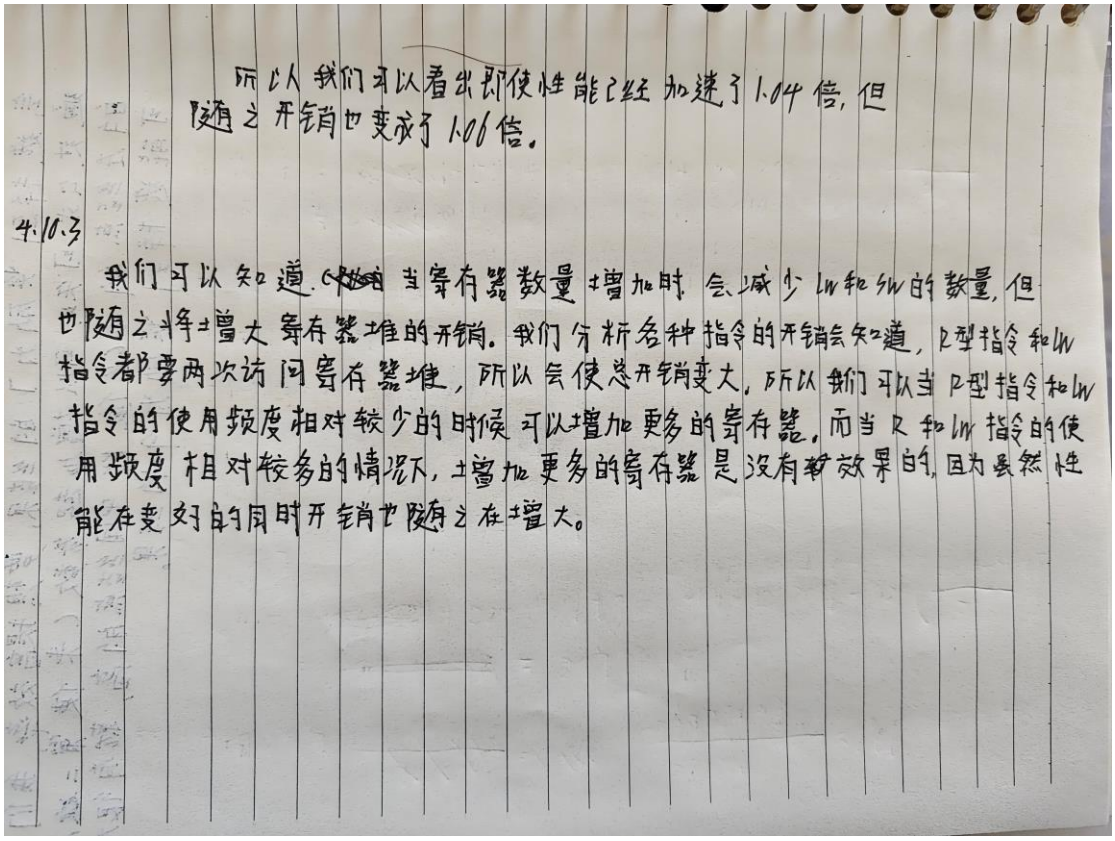
而改进后，因为寄存器堆的开销变成了 400，所以各个指令的开销也要改变。

R 型的开销：2081 + 2 × 200 = 2481
lw 的开销为：4181 + 2 × 200 = 4581
sw 的开销：3961 + 200 = 4161
beq 的开销：1791 + 200 = 2091
I 型的开销：2181 + 2 × 200 = 2581

所以总开销为：

总开销 = 22 × 4581 + 9.68 × 4161 + 12 × 2091 + 52 × ((2481 + 2581) / 2)
= 297764.48

所以开销变化为 297764.48 / 281600 ≈ 1.06


所以我们可以看出即使性能已经加速了1.04倍，但
随之开销也变成了1.06倍。
4.10.3
我们可以知道，当寄存器数量增加时，会减少lw和sw的数量，但
也随之将增大寄存器堆的开销。我们分析各种指令的开销会知道，R型指令和lw
指令都要两次访问寄存器堆，所以会使总开销变大。所以我们可以当R型指令和lw
指令的使用频度相对较少的时候，可以增加更多的寄存器。而当R和lw指令的使
用频度相对较多的情况下，增加更多的寄存器是没有效果的，因为虽然性
能在变好的同时开销也随之在增大。

4.15

4.15

4.15.1

改进后的 lw 指令就不用再使用 ALU，我们可以计算改进后的 lw 指令延迟时间：

$$T(lw) = 30 + 20 + 250 + 50 + 25 + 150 + 20 + 250 + 25 + 150 + 20$$

$$= 990\ ps$$

因为 $T(lw)$ 依然大于 $T(R)$

所以 $T(CPU) = T(lw) = 990\ ps$

No.

4.15.2

更慢，因为改进后 lw 指令就被拆成了 lw 和 add 指令，所以这意味着我们要两次从 PC 获取指令，两次译码指令，四次访问寄存器堆和对 ALU 访问多路选择器，这样显然会使程序的总运行变慢。分析上述，它是比原先慢了：$30 + 50 + 25 + 150 + 20 + 25 + 150 + 20 = 470\ ps$，也就是每个 lw 指令都会变慢 470 ps。

4.15.3

应用程序在新 CPU 上的运行速度主要因素是指令数和指令的 CPI。

4.15.4

我觉得新的 CPU 整体设计更好一点。首先它降低了 lw 指令的时间，也就是把 CPU 的时钟周期给降低了，从而提高了 CPU 的主频。新的 CPU 上没有哪一个指令会同时用到 ALU 和数据寄存器，这也降低了执行指令时的结构复杂度和逻辑模块的负荷。

No.

Date.