Shanghai Jiao Tong University

Final Homework of Computer Organization

Author: *LIU Qiaoan 520030910220*

Course: *Spring 2022, CS2603* – Professor: *Luo Liwen*
Date: *June 17, 2022*

You can see each problem at Problem 1, Problem 3, Problem 4 and Problem 5.

Problem 1

In the textbook [1], we know that there are eight great ideas in computer architecture. In DGEMM (Double precision GEneral Matrix Multiply), we can see that there are at least 3 great ideas exploited. Now we deep into Chapter 3, 4, 5, 6 to discuss it.

- In Chapter 3, we use *subword parallelism* to see how **data-level parallelism** impact the performance of the program. Compared with the unoptimized C code, the Optimized C version of DGEMM using C intrinsics to generate the AVX subword-parallel instructions, which uses `__m256d` data type to store 4 double-precision floating-point values and parallelly executes the inner loop. Figure 1 shows an example.

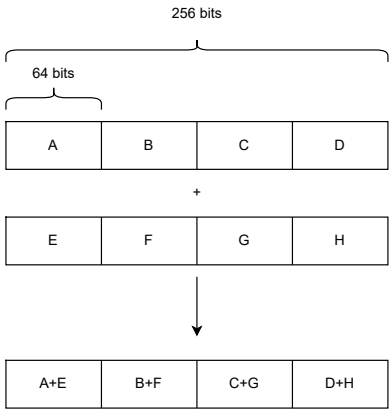


Figure 1: An example of subword parallelism

- In Chapter 4, we unroll the loop to inplement **instruction-level parallelism**. After unrolling loop, there should be more instructions, but the instructions become more independent, which makes it more convenient to inplement instruction-level parallelism. What's more, **pipeline** and **prediction** are also used in DGEMM, since we avoid data hazard and make less stalls, and use prediction to reduce control hazard when meeting jump or branch instruction in loop. Figure 2 shows how unrolling loop works.

- In Chapter 5, we use the **hierachy of memories** to improve the performance. As the section 5.4.5 says, we can divide the matrix into many blocks, then reuse the data in the

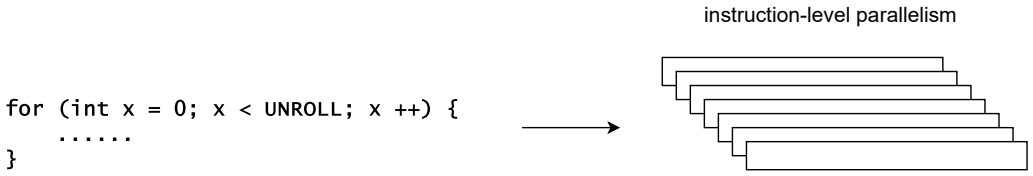


Figure 2: Instruction-level parallelism

cache to lower miss rates due to improved temporal locality. This scheme will not make the program more complex, it just need an extra parameter `BLOCKSIZE` to denote the block size. Figure 3 shows an example.

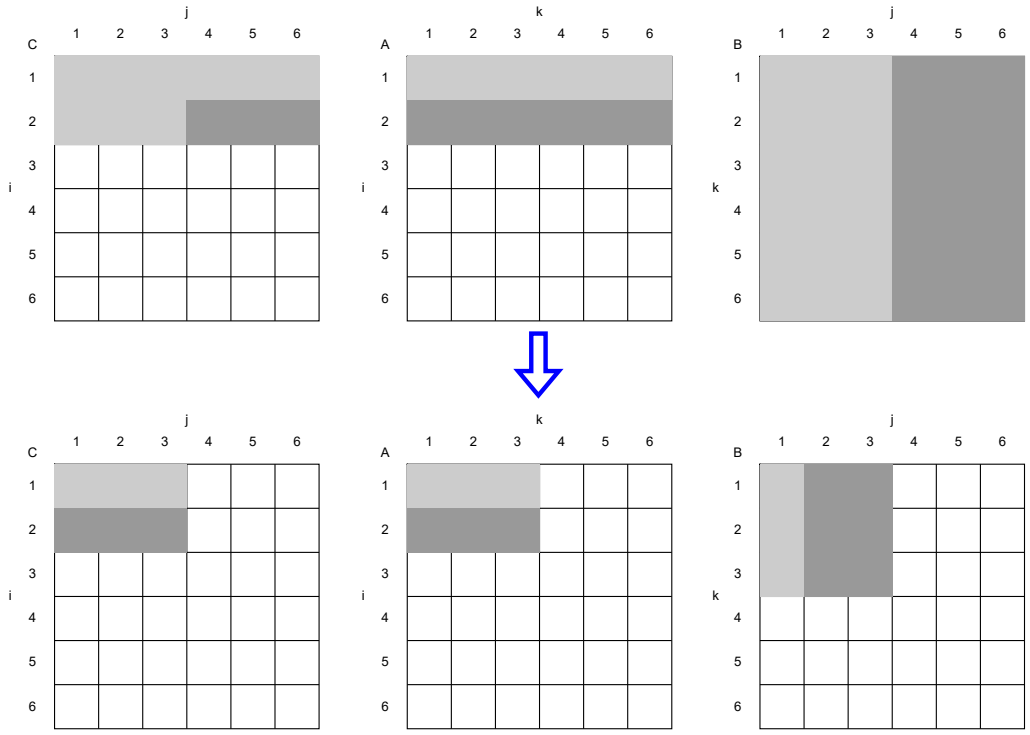


Figure 3: An example of *blocked* algorithm. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. We can see that without blocking, the locality is not well (matrix B will be visited entirely in inner loop), while blocking can make use of locality.

- In Chapter 6, we use multiple processors to improve the performance further. We can spread the work of the outermost loop across all the threads. It's also an example of **parallelism**. Figure 4 shows how the computer spreads the work of the outermost loop across all the threads.

What's more, another 4 great ideas also indirectly impact the performance of DGEMM. They are important ideas for computer architecture.

- **Design for Moore's Law.** As the rapid growth of integrated circuit resources, when we design computer architecture, we must predict where the technology will be. As an example in DGEMM, when Intel began to design AVX (Advance Vector Extensions), it must foresee where the technology will be at 2011. And Intel plans to extend the AVX

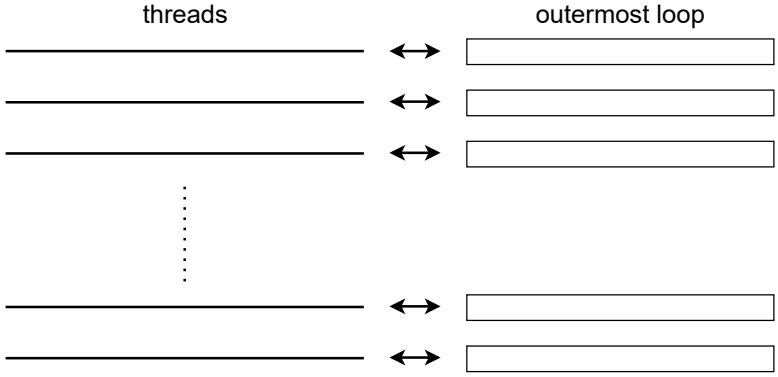


Figure 4: The computer spreads the work of the outermost loop across all the threads.

register to 512 bits, later 1024 bits in later editions of the x86 architecture. This can improve the parallelism in DGEMM more.

- **Use Abstraction to Simplify Design.** In computer architecture, lower-level details are hidden to offer a simpler model at higher levels. As we can see, the hardware in a computer can only execute extremely simple low-level instructions. To go from a complex application to the primitive instructions involves several layers of software that interpret or translate high-level operations into simple computer instructions. We take DGEMM as an example, since programmers have inverted high-level language, we can just write down the C or C++ code for a matrix multiply, with no need to directly write assembly code.

- **Make the Common Case Fast.** This idea is usually implicitly behind the design of computer architecture. As we can see, setting x0 to be always low-level is an example of this idea since we use this value very often. Also, we exploit locality in hierachy of memories because we always visit those data we have just visited or in the neighbor, we must make those operations faster. In DGEMM, we indirectly use this idea to block the matrix into submatrices.

- **Dependability via Redundancy.** Computers not only need to be fast, they need to be dependable. Many physical device can fail, such as disk (in section 5.5). We must use redundancy to ensure that errors can be detected and corrected. The Hamming single error correcting, double error detecting code (SEC/DED) is use widely in memories. Further, CRC code and RS code are also used in network or some other systems. They are effient schemes to make full use of redundant bits to contain more information.

Problem 3

(1) First, we conclude the characteristics of four types of instructions: computational instructions, memory access instructions, control flow instructions and system instructions [2].

- **Computational Instructions.** RV32I comprises 21 computational instructions, including arithmetic, logic, and comparisons.

  - *Arithmetic Operations.* The arithmetic operations are addition, subtraction, and bitwise shifts.
    * The **R-type** instructions ADD and SUB read the source registers *rs1* and *rs2*, then write the result of arithmetic to *rd*. SLL, SRL, and SRA shift the value in *rs1* by the least-significant five bits of register *rs2* and write the result to *rd*. In R-type instruction, the position of these three registers are the same.
    * The **I-type** instructions are similar to the R-type, which uses 12-bit *immediate[11:0]* to replace *func7* and *rs2* in the encoding. Then other parts are aligned with R-type instructions. ADDI, SLLI, SRLI, and SRAI perform the same operation as the counterparts in R-type. Note that SLLI, SRLI, and SRAI only need a 5-bit immediate, since the register only has 32 btis, we only need 5 bits to denote the shift offset. And the remained 7 bits will act as func7. See in Figure 6.
  - *Logical Operations.* The logical operations perform bitwise Boolean operations.
    * The **R-type** instructions AND, OR, and XOR perform their eponymous operations on the values in registers *rs1* and *rs2*, then write the result to *rd*.
    * The **I-type** instructions ANDI, ORI, and XORI do the same, which uses 12-bit *immediate[11:0]* to replace *func7* and *rs2* in the encoding.
  - *Comparison Operations.* The comparison operations perform arithmetic magnitude comparisons.
    * The **R-type** instructions SLT and SLTU perform signed and unsigned less-than comparisons between *rs1* and *rs2*, writing the Boolean result (0 or 1) to register *rd*.
    * The **I-type** instructions SLTI and SLTIU do the same with a 12-bit *immediate[11:0]*.
  - *Two Special Instructions.* There are two special **U-type** instructions LUI and AUIPC. U-type instructions just reserve *rd* and *opcode*, and set the remained 20 bits to *immediate[31:12]*. We can see that U-type instructions are also aligned with other types of instructions.
    * LUI sets the upper 20 bits of register *rd* to the value of the 20-bit immediate, while setting the lower 12 bits of *rd* to zero.
    * AUIPC, short for *add upper immediate to pc*, adds a 20-bit upper immediate to the pc and writes the result to register *rd*.

- **Memory Access Instructions.** RV32I provides five instructions that load a value from memory into an integer register and three that store a value in a register to memory.
  - *Load Instructions.* The load instructions all use the **I-type** instruction format, which use 12-bit *immediate[11:0]* as an offset of *rs1*. Then CPU fetches the result from memery to *rd*. LW, LH and LB load 32-bit, 16-bit and 8-bit quantities, respectively,

and filling the upper bits of *rd* with copies of the sign bit. LHU and LBU are similar, but instead they zero-fill the upper bits.

– *Store Instructions.* The store instructions all use the **S-type** instruction format, which doesn't need *func7* and *rd*. And these two parts are replaced by *immediate[11:5]* and *immediate[4:0]*, respectively. This design ensures that other parts in S-type instructions remain aligned with other types of instructions. The SW instruction copies the 32-bit value in integer register *rs2* to memory. SH and SB copy the low 16-bits and 8-bits in *rs2* to halfword and byte-sized memory locations, respectively.

What's more, we should also notice that there are two types of memory ordering instructions.

– The first is FENCE, which can provide an ordering guarantee between memory accesses prior to the fence and subsequent to the fence. The arguments to the fence are two sets, the *predecessor* set and the *successor* set, which indicate what type of accesses are to be ordered by the fence: memory reads (R), memory writes (W), device input (I), and device output (O). Since there are $2^4 = 16$ types of combination of these four types of access, we need 4 bits to dente the choice of *predecessor* set and the *successor* set, respectively. Then we can see that the format of FENCE instruction is quite special, which is shown in Figure 7.

– Another instruction is FENCE.I, which informs the processor that software has modified instruction memory, so that it can guarantee that instruction fetch will reflect the updated instructions. FENCE.I has fixed binary code shown in Figure 7.

From our class, Prof. Luo taught that these two instructions are quite important, since we need to ensure the coherence of cache.

- **Control Flow Instructions.** RV32I provides six instructions to conditionally change the flow of control (branch instructions) and two unconditional control transfer instructions (jump instructions).

  – *Branch Instructions.* There are six branch instructions, which use the **SB-type** instruction format. This type of instruction divide *func7* and *rd* into four pieces: *imm[12],imm[10:5],imm[4:1],imm[11]*. The other parts are remained the same as R-type. We can see that the encoding scheme of SB-type is quite like S-type, but why is SB-type so confusing? From the class, lab and some material **cite**, we know that *inst[31]* is always used as sign bit to facilitate sign extension, so keep *imm[12]* in *inst[31]* is a good choice to simplify decoding. What's more, only *imm[11](inst[7])* changes role in immediate between S-type and SB-type. See in Figure 8. These six instructions, BEQ, BLT, BLTU, BNE, BGE and BGEU just perform arithmetic comparisons between two registers and can transfer control to anywhere in a range of ±4 KiB (±1K instructions).

  - *Jump Instructions.* There are two jump instructions. JAL is short for *jump and link*, which is a **UJ-type** instruction. This type is quite like the U-type. And we can see that UJ-type is also confusing like SB-type. The reason is similar. We need *inst[31]* as the sign bit to do sign extension. And only *imm[11]* changes role in the immediate.

- **System Instructions.** RV32I has eight system instructions. They are all **I-type** instructions.

  - The ECALL instruction is used to invoke the operating system to perform a system call. Its *opcode* is 1110011, with other bits to be zero.
  - The EBREAK instruction is used to invoke the debugger. Similar to ECALL, their only difference is *inst[20]*.
  - There are six more instructions provided to read and write *control and status registers* (CSRs) [3]. Since they are I-type instructions, there are 12 bits for immediate, and up to $2^{12} = 4096$ CSRs. Presently, this space is only very sparsely populated. However, the upper 4 bits of the CSR address (*csr[11:8]*) are used to encode the read and write accessibility of the CSRs according to privilege level. For CSRRW, CSRRS and CSRRC, they all copy the value in a CSR into integer register *rd*, then CSRRW will write the value in *rs1* into CSR, CSRRS will set the bit if it's set in *rs1* (i.e. use bitwise OR of the CSR and *rs1* to replace CSR) and CSRRC will clear the bit if it's set in *rs1*.

Then we list all the encoding scheme of RV32I.

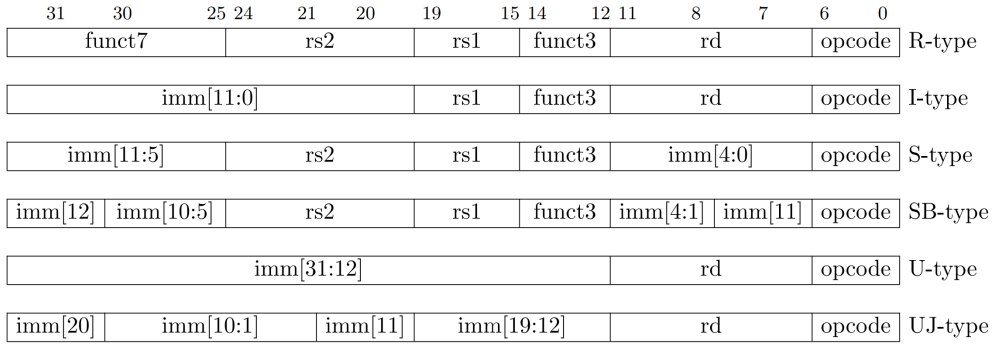


Figure 5: Six types of instructions

And there are special cases for SLLI, SRLI and SRAI. They are all I-type but have only 5 bits for *immediate*.

0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

Figure 6: Special cases for SLLI, SRLI and SRAI

FENCE and FENCE.I are two I-type instructions, but they have special encoding. See in Figure 7.

0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I

Figure 7: FENCE and FENCE.I

S-type and SB-type have the same length of *immediate*, but the encoding scheme of SB-type seems to be confusing. That's because it need to adjust for sign bit and alignment. Similarly, U-type and UJ-type need the same scheme. See in Figure 8.

Figure 8: 32-bit immediate produced by S-type and SB-type instructions

(2) RISC-V is designed for pipeline in many perspectives:

- First of all, all the instructions in RISC-V can be executed within one clock cycle, unlike x86 and CISC whose instruction may need several cycles to execute.
- RISC-V has small size ISA, which means it can be easy to decode within one clock cycle.
- Most instructions in RISC-V have the same length, which means it can be easier to fetch and decode instructions.
- Following the above one, there are few and regular instruction formats, so most instructions in RISC-V are aligned, which means that the decoding process can be done effiently without much expense of hardware.
- For load/store instructions, RISC-V always calculate address in 3^{th} stage then access memory in 4^{th} stage.
- RISC-V makes instruction memory and data memory apart, then instruction fetch and data accessing can be done simultaneously without structural hazard.
- RISC-V uses *bypassing* (*forwarding*) to resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmervisible registers or memory. This significantly reduces stalls and ensure effient pipeline.
- We can use prediction and take hardware (comparer, adder) of branch instructions advance into ID level to reduce the stalls due to control hazard.

Problem 4

First, let's review the problem caused by **power wall**. In the past several decades, as the increase in clock rate, the power grew together. Although power provides a limit to what we can cool, in the post-PC era the really valuable resource is energy. Battery life can trump performance in the personal mobile device, and the architects of warehouse scale computers try to reduce the costs of powering and cooling 100,000 servers as the costs are high at this scale.

The main method to deal with this problem is lowering the voltage, which occurred with each new generation of technology, and power is a function of the voltage squared. However, in 20 years, voltages have gone from 5V to 1V. We cannot exploit this scheme more, since we need base voltage to ensure normal hardware support. So we need other methods to overcome power wall.

Second, we are going to explore two classical solutions in the post-PC era.

- IBM Z is a family name used by IBM for all of its z/Architecture mainframe computers. The technique taken by IBM for implementation on their large Z servers was to **include sophisticated and costly cooling technologies**.
  - IBM z13 and z14 use water-cooling system, and z15 continues to offer a choice of client air cooled (RCU, internal radiator) or client water cooled (WCU) systems for cooling the high-performance processor modules [4]. (The copper tubing feeds cold water to cooling units in direct contact with the CPU chips. Each CPU chip is laid out not to have "hot spots")
  - In 2021, IBM announced its z16 technology. The IBM z16 cooling system is available only with the Radiator (air) cooling option [5]. The previous water-cooling solution that was available with IBM z15 is not offered with IBM z16. This allowed CPU clock rates in excess of 5.0 GHz. The new core IBM Telum dual processor chip in z16 has 16 cores, leverage the density and efficiency of 7nm chip technology, runs at **5.2 GHz**, and delivers increased performance and capacity across a wide range of workloads.
- The technique taken by commodity processor providers, such as Intel and AMD is moving to a **multicore design**, in which each CPU chip contains a moderate number of components. Let's consider the newest product of Intel: Intel® Core™ i7-12650HX Processor, launched in 2022. It has 14 cores and up to 4.70 GHz clock rate. Due to cost constraints, the chips produced by Intel and AMD are cooled by simple fans and all-metal heat radiators.

What's more, with the development of science and technology, more ideas are proposed.

- Intel has been focused on driving new levels of energy-efficienc through **silicon**, **processor**, **platform** and **software** innovation. Intel researchers continue to push the limits of transistor density in next-generation process technologies, while simultaneously driving down power consumption. Intel is also delivering software tools, training and support

that help developers optimize their software for multi-core processors and 64-bit computing. We can see that the results of these efforts are clearly evident in the new products, which dramatically increase performance and energy-efficiency compared to previous generations.

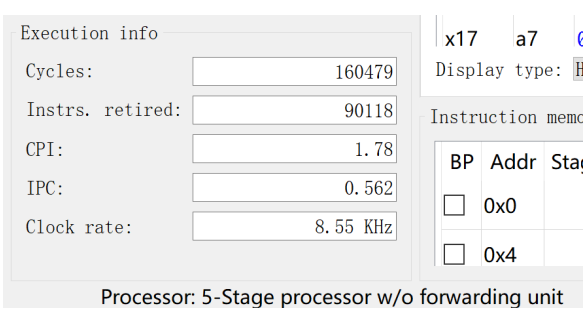
- As a bonus of multicore design, we can increase the size of an on-chip cache memory (The newest Intel® Core™ i7-12650HX Processor has 24 MB cache, while the previous Intel® Core™ i5-12450HX Processor has only 12 M cache). This is exactly one of the most cost-effective and power-effective ways to boost performance.

- GPU also plays an important role in parallelism, which can improve the performance of some special computation, such as miltimedia decoding. This technology also helps to reduce the burden of CPU , makes power lower and performance better.

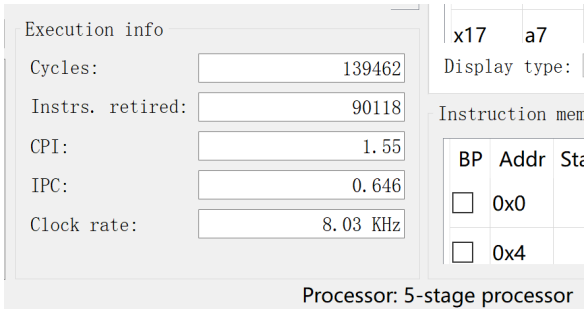
Problem 5

(1) I first write down the cpp file, and then use it to write the assembly code.

(2) I write the first version assembly code, which uses the first element of array as pivot when `divide`. See in Figure 9(a) and 9(b).



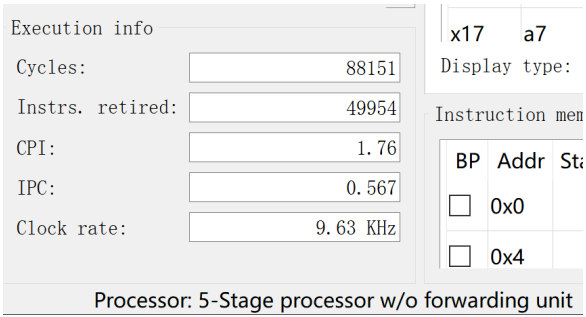
(a) 5-stage processor w/o forwarding unit



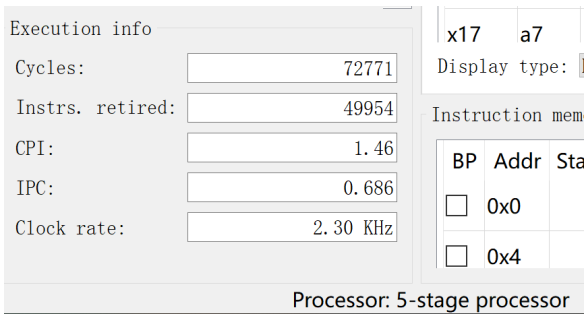
(b) 5-stage processor

Figure 9: The result of first version assembly code

Then I do some optimization, including **pivot choice** and **redundant instructions deletion**. The second version assembly code performs better. See in Figure 10(a) and 10(b).



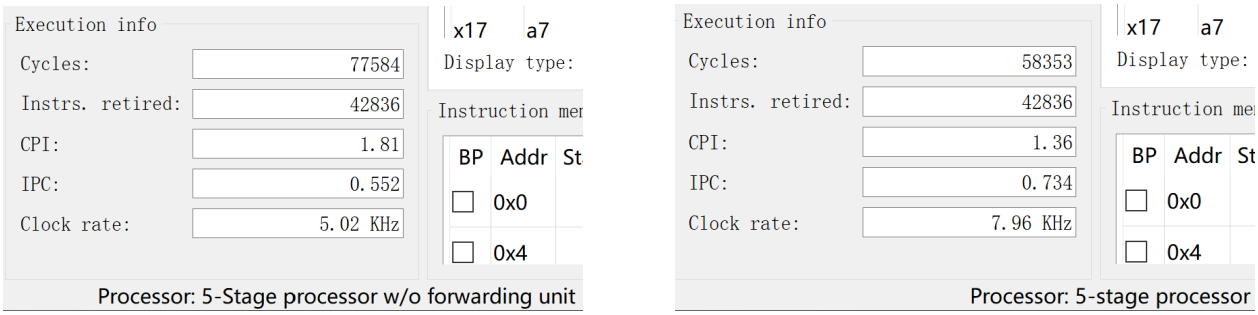
(a) 5-stage processor w/o forwarding unit



(b) 5-stage processor

Figure 10: The result of second version assembly code

After that, I communicated with my classmates, and learned that we can also use **loop unrolling**. Then the third version assembly code performs better. See in Figure 11(a) and 11(b).



(a) 5-stage processor w/o forwarding unit (b) 5-stage processor

Figure 11: The result of third version assembly code

(3) In this section, I test many combination of number of lines, number of ways and size of blocks for the L1 data cache. From our class, we have learned that increase number of ways and size of blocks can lower the miss rate. So I set the number of lines to be 0, then change the number of ways and size of blocks. Note that we only have 128 words for data, so the sum of the three number should be $\log_2 128 = 7$. After testing all the situations, I find that when number of lines to be 0, number of ways to be 2 and size of blocks to be 5, the hit rate is highest. See in Figure 12(a).

We can also see the trend in Figure 12(b).

(a) The highest hit rate (b) The trend

Figure 12: Highest rate and trend

References

[1] John L Hennessy and David A Patterson. Computer organization and design risc-v edition: The hardware software interface, 2017.

[2] Andrew Waterman. Design of the risc-v instruction set architecture. 2016.

[3] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A Patterson, and Krste Asanovic. The risc-v instruction set manual volume 2: Privileged architecture version 1.7. Technical report, University of California at Berkeley Berkeley United States, 2015.

[4] Dustin W. Demetriou, Milnes David, A. Cory VanDeventer, Randy Zoodsma, and Donald W. Porter. Advances in ibm z water cooling: z13, z14, z15. In *2021 20th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (iTherm)*, pages 394–401, 2021.

[5] Ewerson Palacio Makus Ertl Jannie Houlbjerg Hervey Kamga Gerard Laumay Slav Martinksi Kazuhiro Nakajima Martijn Raave Paul Schouten Anna Shugol Andre Spahni John Troy Roman Vogt Bill White, Octavian Lascu and Bo Xu. Ibm z16 technical introduction. Technical report, International Business Machines Corporation, 2022.