LOW-POWER, LOW-COMPLEXITY RISC-V MICROPROCESSOR DESIGN AND IMPLEMENTATION

Zhaoyu Wang

3rd Year Project Final Report

Department of Electronic &
Electrical Engineering

UCL

Supervisor: Prof. Robert Killey

21 March 2022

I have read and understood UCL's and the Department's statements and guidelines concerning plagiarism.

I declare that all material described in this report is my own work except where explicitly and individually indicated in the text. This includes ideas described in the text, figures and computer programs.

This report contains 36 pages (excluding this page and the appendices) and 10261 words.

Signed: Zhaoyu Wang Date: 30/03/2022

Table of contents

Low-power, Low-complexity RISC-V Microprocessor Design and Implementation

Zhaoyu Wang

Microprocessors have already demonstrated high performance in various tasks, such as control systems, sensing, and IoT. However, the power consumption and the complexity are a concern, since they are the significant factors that cause increased cost, environmental impact and, in mobile applications, shorten battery life. For many applications, microprocessors should have a less complex structure while consuming lower power. This report describes a project which aimed to implement a low power and low complexity microprocessor design, based on the RISC-V instruction set architecture. A planned microprocessor design implementing 21 of the 32-bit instructions selected from RV32I which are most commonly used, with clock gating to reduce power consumption and complexity, is presented. It is shown that using this power-saving approach, 13.9% of dynamic power can be saved. Based on this design, a microarchitecture was then designed, corresponding to the 21 chosen instructions, and implemented using the SystemVerilog hardware description language. The report also compares the performance between the single-cycle microprocessor and the pipelined one and a program execution time reduction of 18.7% was achieved.

1 Introduction, Literature Review and Theoretical Background

1.1 Microprocessor

A microprocessor is a digital integrated circuit- it is the processing unit in a computer consisting of arithmetic, logic, and control parts. Circuits implemented these parts are integrated into a small chip and work together to perform arithmetic and logic operations, such as adding, subtracting, comparing numbers, and judging conditions. Both sequential and behavioural logic components are applied to realise the functions above. This digital integrated circuit accepts binary data at its input, processes the information using instructions stored in its memory, and generates the results as outputs in binary form [1]. Four steps are taken in each round of the microprocessor operation- Fetch, Decode, Execute, and Store: the microprocessor fetches the instruction using the current program address from memory, decodes the instruction using a decoder which instructs the microprocessor datapath which tasks should be done, executes to finish the tasks, and stores the results in the memory.

With the establishment and development of the metal oxide semiconductor (MOS) IC technology, and in particular, complementary metal oxide semiconductor (CMOS), there has been a significant increase in the transistor density on a unit chip year by year. This development enabled the computing process to be carried out using multiple MOS chips in the late 1960s. In 1971, Intel produced its first microprocessor design, the Intel 4004 on a single IC. It could only handle 4-bit words. Along with the short representation range of signed numbers (-8 to 7, due to 4-bit words), the clock drives the processor at only a low speed, 108kHz. However, it was the very first generation of microprocessors in history. Fifty years have since passed, and we have witnessed massive progress in microprocessor design. Different types of microprocessors are designed for multiple purposes. For example, the graphic processing unit is a microprocessor with high performance in image rendering.

Nowadays, we can see the implementation of microprocessors in all computer systems. Microprocessors are used in a wide range of applications, from those requiring high performance to those which do not need so much processing but have long battery life. The technology is widely applied to mobile and wearable devices, control systems, communication systems, and numerous IoT applications [2]. To summarise, below are some of the characteristics that make microprocessors advantageous:

- Low manufacturing cost
- Low power consumption
- High portability and compatibility
- Small, compact
- High speed in moving data between different locations of memory

1.2 RISC-V

Many microprocessor architectures have been invented, for example, MIPS, x86, and ARM. However, this report focuses on another architecture, RISC-V, and implements it in a new microarchitecture. RISC-V is an instruction set architecture (ISA) designed by researchers at the University of California, Berkeley. The architecture is designed based on reduced instruction set computer (RISC) principles. The RISC principle is proposed to offset the need to process more instructions by increasing the speed of each instruction, and implementing an instruction pipeline may be simpler if simpler instructions are applied [3]. RISC-V is an entirely open-source ISA, so the payment of a licensing fee is not required when including the RISC-V core in a commercial product [4]. Unlike the other incremental ISAs, RISC-V is developed as a modular instruction set architecture. Therefore, users can choose the optional modules (called extensions) themselves and apply them to the core module instead of implementing new ISA extensions and the previously designed ones [5]. RISC-V consists of a base ISA (RV32I), which is always included, and the other optional standard extensions, which can be included by informing the RISC-V compiler. For example, RV32F is introduced to handle single-precision floating-point, and RV32M can be applied for multiplication. RISC-V instructions are usually 32-bit in length; however, RISC-V supports operation to 64-bit registers with 32-bit instructions in RV64I and enables the compressed instructions with 16-bit length in RVC.

In the core RV32I, instructions are classified into R-Type, I-Type, S/B-Type, and U/J-Type, as shown in figure 1. Despite the different types, they all have a 7-bit opcode. R-type handles two register operands and stores the result in a destination register. I-type handles an immediate number operand and a register operand, and the result can also be found in the destination register. S/B-Type instructions are like I-Type except for replacing the destination register with a 5-bit immediate value. Finally, the U/J-Type instructions consist of only one destination register, a 7-bit opcode, and a 20-bit immediate number [6]. In the RV32I, there are 32 x registers with 32-bit width each, and the register x0 is hardwired with 32-bit zeros [14].

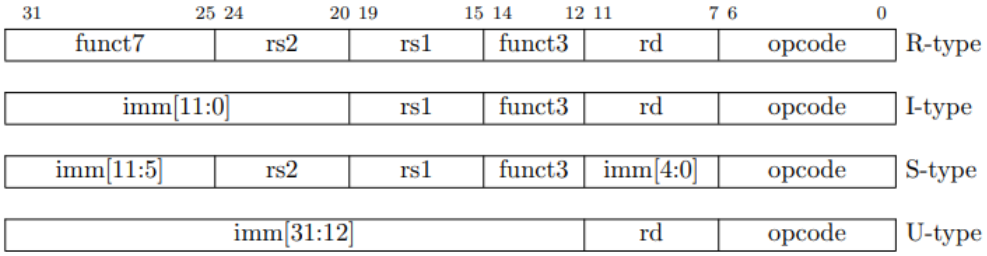


Figure 1, The base instruction formats of RISC-V [6]

The architecture of a computer includes the instruction set and the architectural state, which is the case with the RISC-V architecture. In a RISC-V processor, the architectural state always consists of the program counter and the 32-bit registers. This state is compulsory in all microarchitectures of RISC-V microprocessors. Whenever the microprocessor handles an instruction based on the current architectural state, there is a new architectural state [6]. However, sometimes the non-architectural state is included in the microarchitectures so

that the processor can handle some logic operations more efficiently.

Several advantages make RISC-V a good choice:

• Simple architecture, easy to understand

• High portability

• Modular ISA

• Complete toolchain

• Open-source ISA

In addition, in many RISC-V microprocessors, the addresses are word-aligned, which means that the word address should be divisible by 4 (as each 32-bit word is made up of 4 bytes) [6]. The following address can be found by adding 4 to the current word address, and the elements in the memory next to each other have a difference in address by 4.

1.3 Performance and Power Consumption

The performance and power consumption of the microprocessor should always be kept in mind. The most direct way to measure performance is to find the execution time of a test program or the total execution time of a collection of programs [6]. The execution time (in seconds) of a program is given as equation 1.1:

$$ExecutionTime = (\#instructions)\left(\frac{cycles}{instruction}\right)\left(\frac{seconds}{cycle}\right) \quad (1.1)$$

The first term at the right-hand side of equation 1.1 stands for the number of instructions in the test program, which might be different due to the different architectures applied (for example, RISC and CISC) and the way the programmer writes the program. The second term shows the number of clock cycles required to finish a single instruction. The third term is the time in seconds taken per clock cycle, which defines the clock period.

One of the parameters that can measure the microprocessor's performance is the Cycles-per-instruction (CPI). As the name suggests, it represents the number of clock cycles taken to finish one specific instruction. The reciprocal of CPI, the instruction-per-cycle (IPC) stands for the number of instructions that have been executed in a clock cycle, giving a direct measure of the processor's throughput.

Another factor that determines how the microprocessor performs is the clock period. The clock signal is used in the microprocessor to drive the time-sensitive elements in the processor, such as the register file, the PC counter, and the memory. The clock period depends on the critical path, the slowest path, which limits the speed of the circuit from input to output through the logic in the processor [6]. Different microarchitectures have different circuit layouts, hence the different clock periods. Modifying the logic and circuit designs could affect the clock length of the processor [6].

However, there is a trade-off between performance, power consumption, and manufacturing cost. Choosing the design that could significantly reduce the execution time while satisfying cost and power consumption constraints can be challenging [6]. Different microarchitectures of the processor are always compared, and modifications are always made to optimise the design since both factors mentioned above are dependent on the microarchitecture.

Power consumption is also an essential feature of a microprocessor. Due to the increasing number of transistors in a single microprocessor, much power is consumed and converted into heat energy, reducing the processor's stability and performance [7]. There are two sources of power consumption: dynamic and static power consumption. [8]

- Dynamic power consumption
  From the term "dynamic", this consumption occurs when there are activity changes in the circuit, such as in the input and clock signals. There are two sources: the switched capacitance, a primary source that arises from the charging and discharging of capacitors at the outputs of the circuit, and the short-circuit current, a secondary source that flows simultaneously when the CMOS gates switch [8] due to the property of transistors. Compared with the switched capacitance, the power consumption from the short-circuit current takes a smaller percentage, and more focus is paid to minimising power dissipation in switched capacitance.
  The dynamic power from the switched capacitance is the predominant dissipation form and is described by:

$$p = \alpha C V^2 f \qquad (1.2)$$

  for each node, where α stands for the switching activity, C is the load capacitance, V is the voltage applied, and f stands for the clock frequency [9].

  Four approaches have been proposed to reduce dynamic power consumption, according to the formula 1.2. First of all, the microprocessor is designed to minimise switching activity. Since we cannot stop the input and output from changing values, we can prevent the unnecessary switching of the clock signal by using the clock gating technique, which is the method applied in this project. This is an efficient way to save power and energy since the clock network accounts for a significant fraction of a processor's energy consumption [8]. According to the proposed measurement,

clock gating could theoretically save up to 27% of energy usage, according to the result of previous research [10]. The principle for clock gating is also quite simple: the clock which drives the digital circuit comes from the output of an AND gate whose inputs are the clock for the system and the ENABLE signal (which shows if the function unit works at the clock edges). When the ENABLE signal is asserted, the output clock is transparent to the system clock. In the other case, where the ENABLE signal is zero, the clock that drives the idle function block is gated, without introducing switches of the clock at this time.

The second approach is to reduce the capacitance of the circuit. However, according to the research literature, there is always a trade-off between power consumption and performance [8]. For example, the smaller size of the transistor might reduce not only the capacitance but also the processor's performance.

The third and fourth ways deduced from the equation above are reducing the supply voltage and the clock frequency. However, reducing supply voltage increases the gate delay in the transistors and requires a lower clock frequency to match the increasing delay [8]. Applying a lower clock rate could reduce the power consumption, but it will not affect the energy consumption since the speed of the processor is reduced with the clock rate, and the time taken to finish a fixed-length program will be longer. This approach will not save energy but will reduce the processor's performance.

- Static Power Consumption

  The static power, which dissipates in the computer's components, is also known as idle power or leakage [8]. The leakage power occurs when the devices are active, but the signals stay constant [10]. There is always a leakage current I_{leak}, due to the nature of the CMOS transistors. The current leaks when the gate voltage is below the threshold voltage of a transistor because of the transistor's imperfect performance. Since the leakage power is static, it can be found by the product of the supply voltage and the leakage current, and the equation below describes the leakage power [8]:

$$P_{leak} = VI_{leak}$$

1.4 Single-cycle Microarchitecture

The first proposed microarchitecture in this project is a single-cycle design. The single-cycle design is the combination of four state elements (register file, data memory, instruction memory, and PC counter) and the logic blocks for arithmetic or logical operations. Each instruction is fetched, decoded, and executed in the same clock cycle. The corresponding result is stored in the register file or the external memory once the execution result is completed. The microprocessor can handle one instruction in each clock cycle (i.e., CPI=1).

1.5 Pipelined Microarchitecture

The more advanced microarchitecture, the pipelined microarchitecture, is proposed based on the single-cycle design. The processor is divided into five stages: the Fetch stage, which fetches the instruction from memory; the Decode stage, which extracts the

information from the 32-bit instructions; the Execute stage, which performs logical or arithmetic operations; the Memory stage, which stores the result into the data memory if needed; the Writeback stage which writes the result back to specified register in the register file. Two changes are made to the single-cycle design: firstly, the addition of four 32-bit registers, which split the microprocessor into five stages in both the datapath and control unit, and, secondly, a hazard unit in the datapath.

The hazard unit is used to solve the conflicts between the instructions, referred to as hazards. Hazards can be classified into data hazards and control hazards. Data hazards always occur when the destination register (register which stores the result) in the Memory or Writeback stage is the same as a source register (the operand) in the Execute stage before the latest register value is written back to the register file. In addition, using the register which is not prepared in the load instruction could also lead to the data hazard. Control hazards arise at the branches, where the next instruction is not the following instruction in the instruction memory. Thus, the next instruction will not be the expected one with the correct address, and the processor will fetch the following instruction instead without the hazard unit.

1.6 How does the computer program execute?

The figure below illustrates the process from high-level language to machine code.

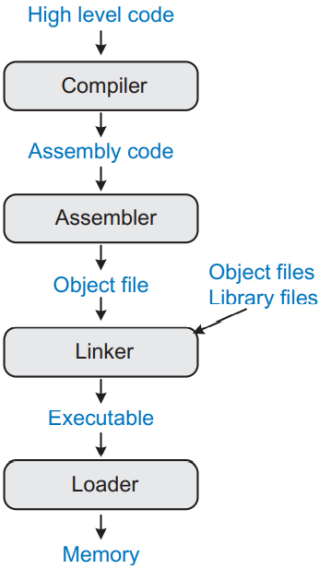


Figure 2, Steps taken for translating and starting program [6]

High-level code such as C and Java is first compiled to obtain the assembly code. This low-level language keeps the strong correspondence between the instructions in the high-level language and the machine code by a compiler [12]. The assembler then translates the assembly code into machine code and puts it into an object file [6]. The following step in the linker is to put the machine codes in the object files and libraries together and assign the corresponding branch addresses and variable locations. The steps mentioned above are usually taken by a compiler, which consists of phases that sequentially analyse given forms of a program and synthesise new ones in series, producing a relocatable object module that can be further processed to link with others and loaded into machine

language [13]. The loader loads the program into memory in the final stage, and the processor starts working [6].

2 Goals and objectives

There are several goals in the project:

- Implement a variety of microprocessor structures and carry out a comparison of their performance
  - Design single-cycle microprocessor and pipelined microprocessor
  - The two types of microprocessors should be experimentally implemented using a field-programmable gate array (FPGA) development board to verify the performance
  - Compare the two types of structures in throughput, cycle-per-instruction (CPI) to ascertain their properties
- The power consumption should be reduced
  - Power-saving techniques should be investigated and applied to the design
  - The saved power should be proved using sufficient evidence such as calculations and simulation results
- The design should have low complexity
  - The instructions should be selected so that unnecessary instructions will be removed, and the microprocessor only supports the frequently used instructions.

3 Method

The most effective way to design, ensuring the correct system operation is a top-down approach, while the highest level in computer architecture deals with processors, memories, and switches [11]. The microprocessor is designed based on the Three -Y's rules [6]: the hierarchy which involves dividing a system into modules then further subdividing each of the modules until each piece is easy to understand, modularity which consists in assigning each module well-defined functions and interfaces to improve the connectivity, and the regularity which seeks uniformity among modules and reuses modules in common so that the number of the distinct modules can be reduced.

Depending on the goals and objectives for this project, the steps below are taken in the microprocessor design and implementation.

- Processor Design
  - Instruction Selection
    There are over 30 instructions in the base RV32I instruction set. However, the compiling results of standard computer programs in a high-level language might not include all of those instructions. To reduce the complexity, necessary instructions which always occur in the computer programs are selected.
  - The stage elements and logical/arithmetic function block design and verification
    These components make up a whole microprocessor, and it will be easier to design each of the digital building blocks before directly building a large, unstructured microprocessor.
  - Single-Cycle Microarchitecture Design

The Datapath and control unit are designed based on the selected instructions. The datapath, control unit, and external memories are integrated into the single-cycle design.

- Pipelined Microarchitecture Design
  A similar design process as the single-cycle processor design. In addition, the hazard unit is used to tackle the conflicts in some instructions. The datapath, which includes the hazard unit, the control unit, and the external memories, are integrated into the pipelined design.
- System Verilog code realisation and simulation for both single-cycle and pipelined microprocessors
  A software from Intel, Modelsim, can simulate the digital design and sketch the time sequence diagram to check the processor's functionality and behaviour in ideal cases (e.g., ignore the propagation delay of the digital elements, only consider the digital performances).

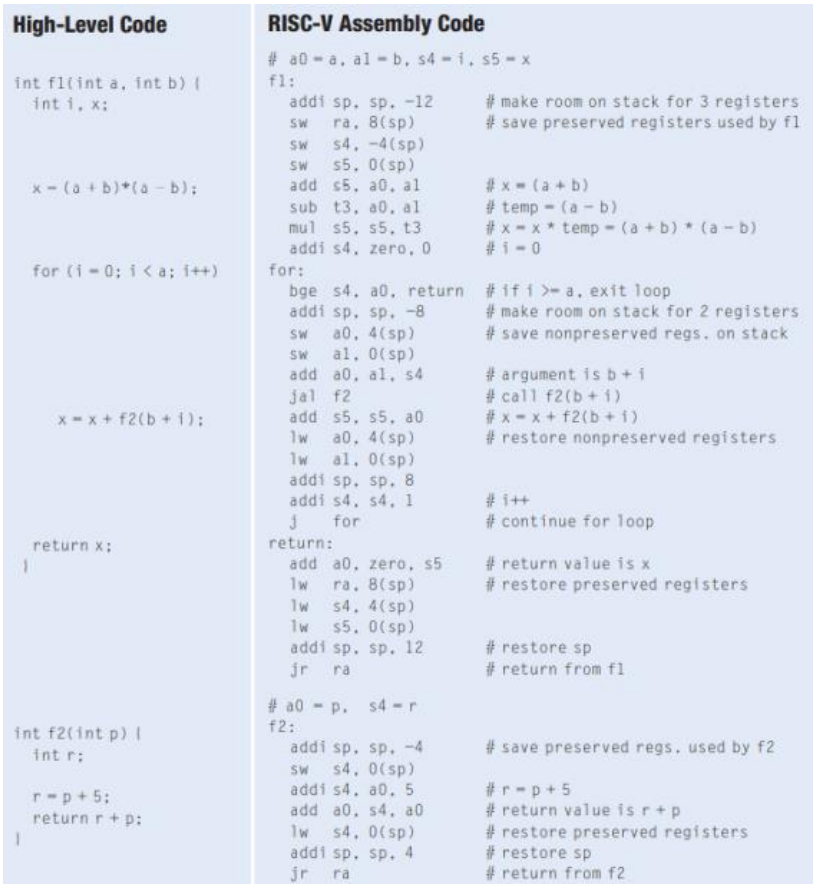
- Implementation of both structures to execute a high-level language code
  In this task, the assembly code interprets a program that computes the first seven numbers in the Fibonacci Series, and the obtained assembly code is translated into machine code using Python. The designs are implemented to execute the program. The microprocessors are updated to the FPGA board, and the peripherals can reflect the execution results on the FPGA board. In this project, each of the eight LEDs on the FPGA board represents one bit of the processor output so that the output values can be checked with what we expect in each instruction to verify the design.

- Microprocessor performance and power consumption analysis
  In this section, the report will compare the performance and power consumption between the two proposed microarchitectures. Several assumptions will be made, such as the number of instructions and the transistors' parameters in the low-level design of the building blocks. The additional power consumption from the hazard unit and the registers in the pipelined design will be estimated. The power consumption for the two processors to execute the same number of instructions are compared to see the trade-off between the power consumption and performance.

4 Result and Analysis

4.1 Instruction Selection

Since it is not necessary to include the complete instruction set, selecting critical instructions becomes the first step, and the selection could affect the actual behaviour of the microprocessor. The architecture after instruction selection, we term the E-RISC architecture, whose complexity has been significantly reduced. While considering which instructions must be included, it is well worth thinking about what a typical computer program will do and what the action of the microprocessor corresponds to when executing that particular program. A code example is shown below in figure 3:



High-Level Code

```
int f1(int a, int b) {
  int i, x;

  x = (a + b)*(a - b);

  for (i = 0; i < a; i++)

    x = x + f2(b + i);

  return x;
}

int f2(int p) {
  int r;

  r = p + 5;
  return r + p;
}
```

RISC-V Assembly Code

```
# a0 = a, a1 = b, s4 = i, s5 = x
f1:
   addi sp, sp, -12      # make room on stack for 3 registers
   sw   ra, 8(sp)        # save preserved registers used by f1
   sw   s4, -4(sp)
   sw   s5, 0(sp)
   add  s5, a0, a1       # x = (a + b)
   sub  t3, a0, a1       # temp = (a - b)
   mul  s5, s5, t3       # x = x * temp = (a + b) * (a - b)
   addi s4, zero, 0      # i = 0
for:
   bge  s4, a0, return   # if i >= a, exit loop
   addi sp, sp, -8       # make room on stack for 2 registers
   sw   a0, 4(sp)        # save nonpreserved regs. on stack
   sw   a1, 0(sp)
   add  a0, a1, s4       # argument is b + i
   jal  f2               # call f2(b + i)
   add  s5, s5, a0       # x = x + f2(b + i)
   lw   a0, 4(sp)        # restore nonpreserved registers
   lw   a1, 0(sp)
   addi sp, sp, 8
   addi s4, s4, 1        # i++
   j    for              # continue for loop
return:
   add  a0, zero, s5     # return value is x
   lw   ra, 8(sp)        # restore preserved registers
   lw   s4, 4(sp)
   lw   s5, 0(sp)
   addi sp, sp, 12       # restore sp
   jr   ra               # return from f1

# a0 = p,  s4 = r
f2:
   addi sp, sp, -4       # save preserved regs. used by f2
   sw   s4, 0(sp)
   addi s4, a0, 5        # r = p + 5
   add  a0, s4, a0       # return value is r + p
   lw   s4, 0(sp)        # restore preserved registers
   addi sp, sp, 4        # restore sp
   jr   ra               # return from f2
```

Figure 3, An example of computer code and corresponding instruction [6]

In a program written with high-level code such as C, C++, and Java, logic and arithmetical operations are always carried out on the variables. Looking deeper into the processor's execution, the operations performed include

- loading the values from registers or memory,
- performing calculations on the loaded data, and
- storing the results back to registers or memory.
- The loop might be required to repeat an action while a specific condition is satisfied; the program may decide what to do next depending on whether certain conditions are met or not;
- functions may be declared in the program to reuse the code, improve code readability, and reduce programming complexity.

21 instructions are chosen to be implemented in the microprocessor design to satisfy the high-level language code:

- R-Type instructions:
  1. Shift left logical $sll\ rd, rs1, rs2$

     Shift left logical is an R-Type instruction that shifts the source register $rs1$ value by n (represented by $rs2$) bits. The result is stored in the destination register rd. A left shift is equivalent to the product with a multiple of 2.
  2. Shift right logical $srl\ rd, rs1, rs2$

Similar format to shift left logical instruction, but the source register value is shifted to the right logically instead. A right shift is equivalent to division by the exponential of 2.

3. Shift right arithmetic $sra\ rd, rs1, rs2$

Similar format to shift left logic instruction, but the source register value is shifted to the right arithmetically. A right shift is equivalent to division by the exponential of 2.

4. Add $add\ rd, rs1, rs2$

Add (add) is an R-Type instruction that adds the two source registers and stores the result to the destination register rd.

5. Subtract $sub\ rd, rs1, rs2$

It is similar to add instruction, but it subtracts the two source register values.

6. Xor $xor\ rd, rs1, rs2$

Similar to add instruction, it finds the bitwise "exclusive or" logic between the two source register values instead.

7. Or $or\ rd, rs1, rs2$

It is similar to add, but it finds the bitwise "or" logic between the two source register values.

8. And $add\ rd, rs1, rs2$

It is similar to add instruction, but it finds the bitwise "and" logic between the two source register values.

- I-Type instructions:

1. Load word $lw\ rd, imm(rs1)$

Load word instruction (lw) is an I-Type instruction that operates two registers and an immediate number. It adds the immediate number offset to the source register, which stores the base address and extracts the data from the resulting address to load the word.

2. Shift left logical immediate $slli\ rd, rs1, uimm$

Shift left logical immediate (slli) is an I-Type instruction that shifts the source register $rs1$ value by n (represented by the unsigned immediate number) bits and stores the result into the destination register rd. A left shift is equivalent to multiplication by a multiple of 2.

3. Shift right logical immediate $srli\ rd, rs1, uimm$

Similar format to shift left logical immediate, but the source register value is shifted to the right logically. A right shift is equivalent to division by the exponential of 2.

4. Shift right arithmetic immediate $srai\ rd, rs1, uimm$

Similar format to shift right logical immediate, but the source register value is shifted to the right arithmetically. A right shift is equivalent to division by the exponential of 2.

5. Add immediate $addi\ rd, rs1, imm$

Add immediate (addi) is an I-Type instruction that adds the source register with the immediate number stores the result to the destination register specified in the instruction.

6. Xor immediate $xori\ rd, rs1, imm$

Similar to add immediate instruction. However, it finds the bitwise "exclusive or" logic between the source register and the immediate number instead.

7. Or immediate $ori\ rd, rs1, imm$

Similar to add immediate instruction. However, it finds the bitwise "or" logic between the source register and the immediate number instead.

8. And immediate $andi\ rd, rs1, imm$

Similar to add immediate instruction. However, it finds the bitwise "and" logic between the source register and the immediate number instead.

9. Jump and link register $jalr\ rd, rs1, imm$

Jump and link register is an I-Type instruction that calculates the program address by adding a value stored in a register with an immediate, writes the address to the program counter and stores the address next to the current one in another register (return address register ra by default) This instruction is suitable when returning from a called function to the caller function.

- S-Type instructions:

1. Save word $sw\ rs2, imm(rs1)$

Save word instruction (sw) is an S-Type instruction that manipulates two registers and an immediate number. First, it adds the source register $rs1$, which stores the base address and the immediate number offset to find the target address and stores the data from register $rs2$ into that memory address.

- B-Type instructions:

1. Branch if equal/not equal $beq\ rs1, rs2, label;\ bne\ rs1, rs2, label$

These are B-Type instructions that operate with two registers. It judges whether the two register values satisfy certain conditions (if they are equal /not equal) and jumps to the labelled instruction. These instructions can be used in conditional branching and the loop condition.

- J-Type instructions:

1. Jump and link $jal\ rd, label$

Jump and link (jal) is a J-Type instruction that writes the address for the next instruction to the labelled one by adding the current address with an immediate number and stores the following address to the current one in the specified register rd (return address register ra by default). Jump and link instruction can be used in conditional branching and loop to execute particular code.

4.2 Single-Cycle Microarchitecture Design

After selecting the necessary instructions from the whole instruction set, the state elements and the function blocks which could realise the instructions should be investigated and combined into the single-cycle design. According to the execution each instruction performs, the corresponding single-cycle microarchitecture is shown in the figure below:

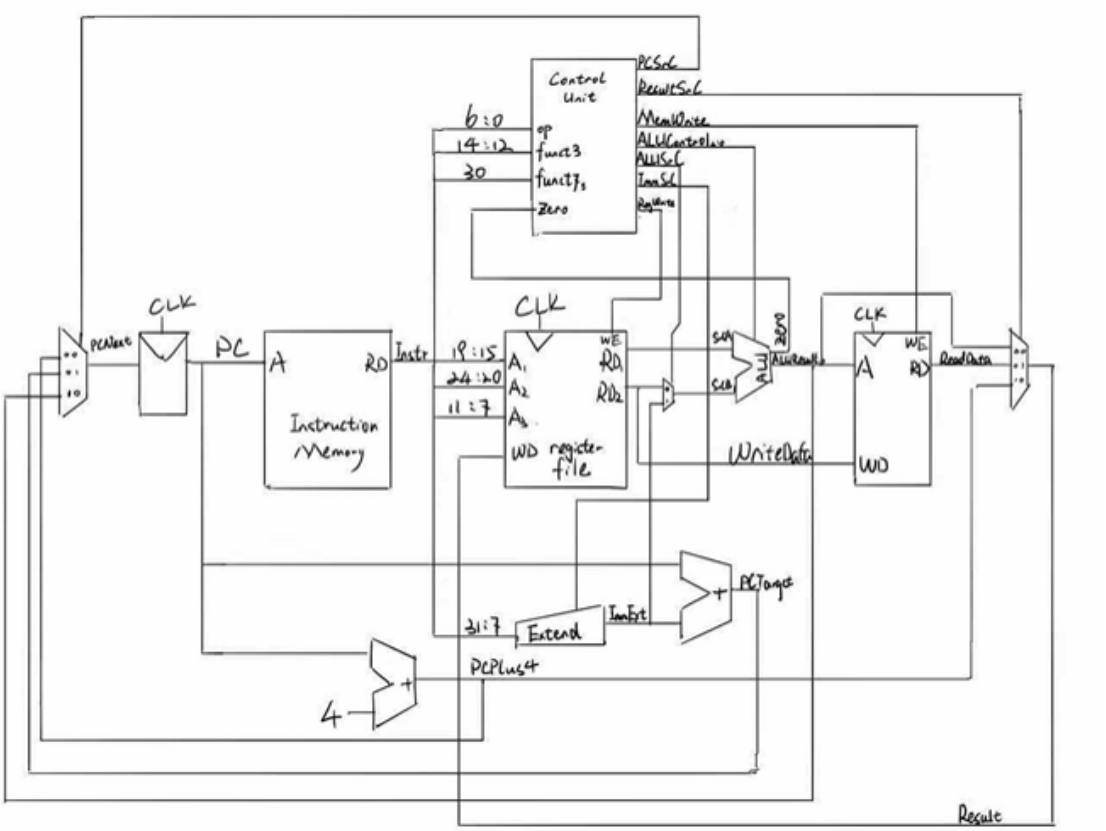


Figure 4, the single-cycle microarchitecture design

The ALU unit is implemented in the microprocessor to deal with both logical and arithmetic operations. The following program address at the program counter, PC, can be either PC+4, the branch address, or the result from the ALU. Thus, the adders are applied to calculate two possible PC addresses, and all three possible addresses for the next instruction are fed to a 3-to-1 multiplexer. The following program memory, PCNext, is then updated at the PC counter, a 32-bit register that triggers at the positive edge of the clock. The instruction memory then reads the output from the PC counter and fetch the instruction corresponding to that address. The instruction at the memory's output is then decoded into different segments, which are fed into the control unit, register file, and extend unit to compute the control signal, the register values, and the extended immediate number with 32-bit width. A 2-to-1 multiplexer chooses the input from either the register file or the extend unit to decide whether the ALU deal with the immediate number or the register based on the instruction type since R-type instruction copes with 2 registers and other instructions might use the immediate number as one of the inputs. The data memory always reads the data at the input address, while it only writes the data at the rise of the clock. Finally, another 3-to-1 multiplexer receives the inputs from the ALU (For R-type and I-type instructions), data memory (for load word instruction) and the address of the following instruction in the memory (for J-Type instruction).

4.3 Pipelined Microarchitecture Design

There are only two changes from the single-cycle microarchitecture to the pipelined design: the addition of registers that split the processor into five stages and the hazard

unit which solves the hazards efficiently. The layout of the pipelined microarchitecture is displayed in figure 5:

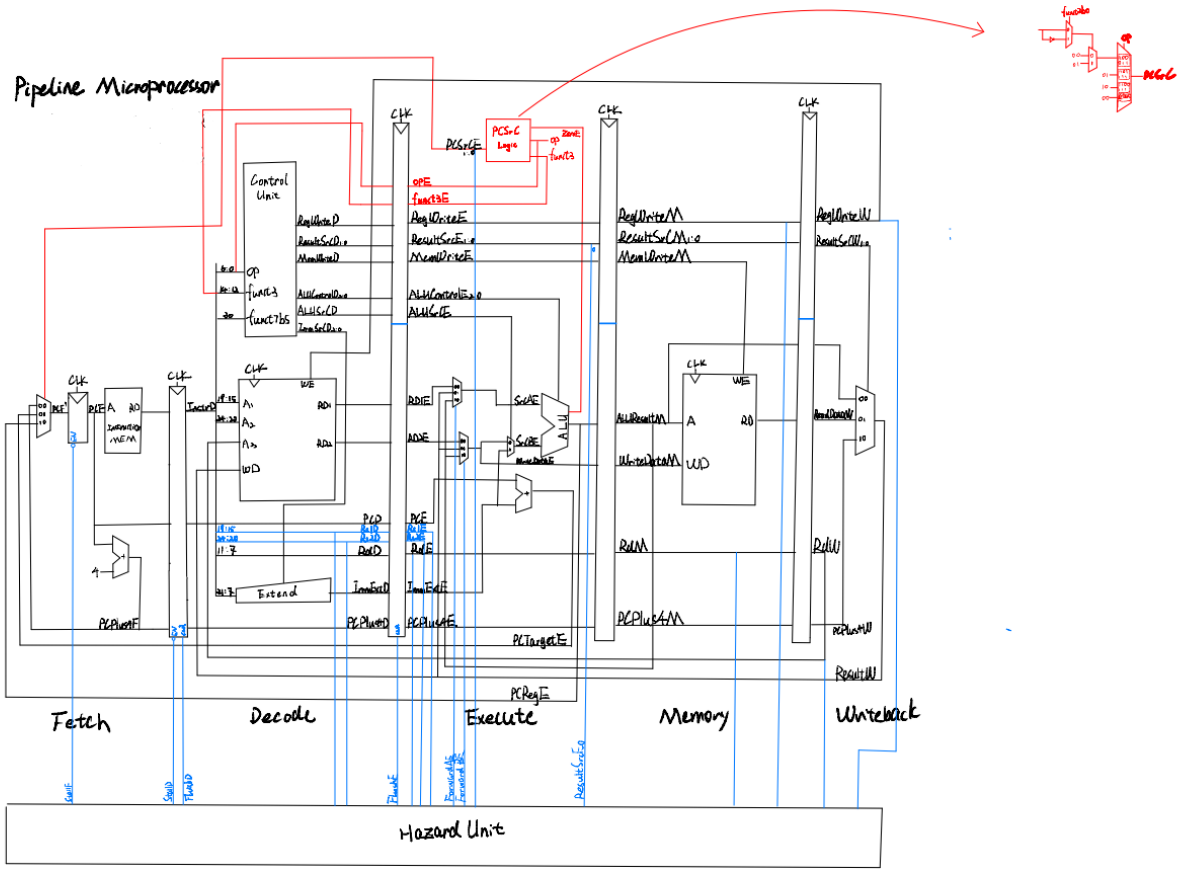


Figure 5, the pipelined microarchitecture design

The function block *PCSrc logic* in red represents a small change in the control unit. Since the decision for the branch instructions is taken in the execution stage from the ALU, the logic to choose the next program address is set in the execution stage in order to keep the system synchronous. The registers between each stage only trigger at the rising edge of the clock so that the 5 stages do their own work on 5 respective instructions at the same clock cycle. Once the instruction has been completed in the current stage, the information of this instruction obtained in this stage will be pushed into the next stage by the register.

There are two sources of potential hazard, the data hazard and control hazard. Three methods are designed in order to solve the hazards:

1. Forwarding

   The forwarding could solve the hazard when the operand(s) in the Execute stage and the destination register in the Memory or the Writeback stage shares the same register. Since the register value has not been updated to the register file in the latter two stages, this register is not the latest in the Execute stage and does not contain the correct data. As a result, two multiplexers are introduced at the input of the ALU, and the hazard unit receives source registers in the Execute stage and the destination registers in both Memory and Writeback stages. Both source registers in the Execute stage can be either from the register file or the destination register from the

Memory/Writeback stages, depending on the control signal from the hazard unit by using the 3-to-1 multiplexers.

2. Stalls
   In the load word instruction, the data should be extracted from the memory before using. However, the data is read at the end of the Memory stage and cannot be forwarded to the Execute stage that deals with the next instruction, introducing a two-cycle latency [6]. By holding the processor, the program keeps going until the data from the memory is ready to be used.

3. Branch prediction
   From the microarchitecture diagram in figure 5, the result whether the branch is taken or not is available in the Execute stage. However, simply stalling the processor for two cycles until the Execute stage finishes will reduces its performance if there are a lot of B-type instructions. The way to solve this problem in this project is to keep executing the following instructions without stalling by predicting the branch is not taken. Once the Execute stage is finished and the branch should be taken, the processor will flush the Fetch stage and Decode stage and start fetching the instruction from the branch address instead. Otherwise, the processor will keep dealing with the instructions in order.

4.4 The stage elements and logical/arithmetic function block design and simulation

The components in both microarchitectures are designed and tested first in order to reduce the design complexity. According to the 3 -Y's rules, the regularity rule should be followed to reduce the number of distinct modules by reusing the modules in common. The functional blocks which are shared by both microarchitectures are designed and simulated. The result is summarised and shown below:

1. Extend unit
   The extend unit extends the immediate number which is included in the instruction into 32 bits with the most significant bit (sign-extended) or zeros (zero-extended). The table below explains how the extend unit extends the immediate number based on the control signal.

Control signal	Instruction type	Extended number
000	I	{{20{Instr[31]}}, Instr[31:20]}
001	S	{{20{Instr[31]}}, Instr[31:25], Instr[11:7]}
010	B	{{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0}
011	J	{{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0}
100	I	{{20{0}}, Instr[31:20]}
101	S	{{20{0}}, Instr[31:25], Instr[11:7]}

110	B	{{20{0}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0}
111	J	{{12{0}}, instr[19:12], instr[20], instr[30:21], 1'b0}

Table 1, Truth Table for the Extend Unit

The simulation result for the extend unit in figure 6 proves that the building block works as expected.

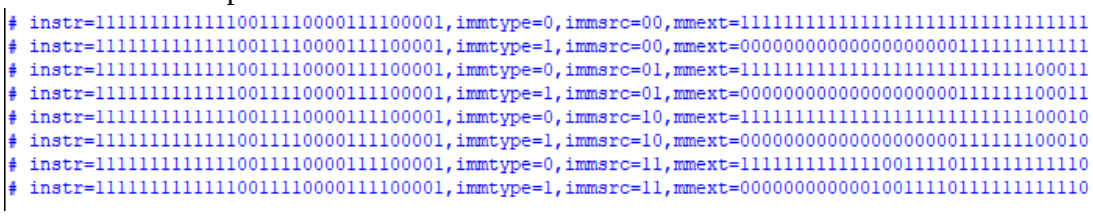


Figure 6, the simulation result for the extend unit

2. Adder

   The adder performs addition between the two inputs and shows the result at the output. In the testbench, different input combinations are tested, and from the result below, the adder works as expected.

Figure 7, the simulation result for the adder

3. Program counter

   The program counter is a 32-bit register that could be reset at the positive edge of the clock signal and when the reset signal is asserted. At the beginning of the test, reset is set to 1 and the output q is initialized to 0. While the reset signal is 0 and the clock signal propagates to 1, there is a rising edge at the clock and the value in the input d is updated to output q, as a result, q is 1. In the later stages, reset is set to 1 again and the output q is set to 0 again. The program counter works as expected.

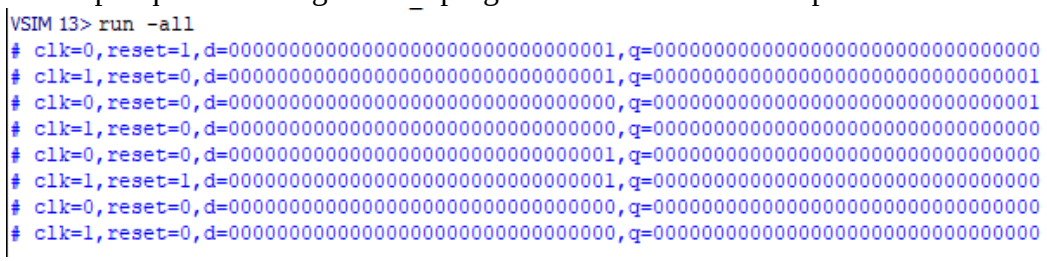


Figure 8, the simulation result for the program counter

4. Register file

   The register file is a clock-sensitive block that could store, read, and write the 32 registers. Two input ports specify the two registers that are read at the two outputs. Another two input ports decide the data input and which register is written at the positive edge of the clock.

   The testbench is also used to test the functionality of the register file. At the beginning, the two read ports read the value at register 0 and output the result. 32-bit 0101 is also written into register 1. The data in register 1 is read after 5 seconds, and the value shows 32-bit 0101. The zero register should remain unchanged, so 01011 is written to the zero register x0. After 5 seconds, the data stored in the zero register x0 is displayed at RD1, which is still 0. 01000 is also written to register 2, which is then

read after 5 seconds. The reading is 01000. As a result, the register file works as expected.

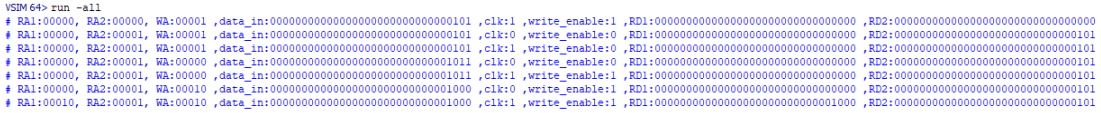


Figure 9, the simulation result for the register file

5. 2-to-1 multiplexer

The multiplexer selects which input will be passed to output by the “select” signal s. By executing the testbench, the “select” signal in the testbench is set to 0 and 1 respectively, and the output y is d0 when s equals 0 and d1 when s equals 1. The multiplexer works as expected.

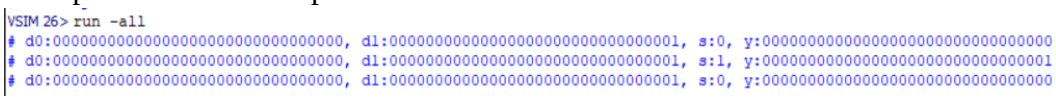


Figure 10, the simulation result for the 2-to-1 multiplexer

6. ALU

The ALU carries out both logical and arithmetic operations between two 32-bit inputs, depending on the 3-bit control signal. The table below demonstrates how the ALU works.

$ALUControl_{2:0}$	Operation
000	a+b
001	a-b
010	a&b
011	a|b
100	a^b
101	a<<b
110	a>>b
111	a>>>b

Table 2, the ALU operation list

The testbench verifies the functionality of the ALU by applying all operations the ALU could take. All of the 8 operations in table 2 are taken by varying the control signal and the output are compared with the expected value. The simulation result show that the ALU works as expected.

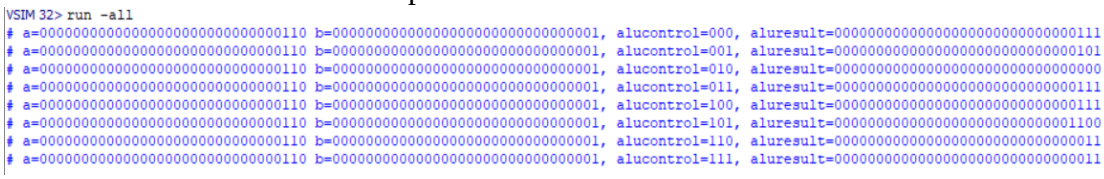


Figure 11, the simulation result of the ALU

7. 3-to-1 multiplexer

It is a similar design to the 2-to-1 multiplexer, but it has 3 inputs and a 2-bit select signal instead. From the simulation result, the output has the value of d0 when s equals 00, the value of d1 when s equals 01, and the value of d2 when s equals 10. The 3-to-1 multiplexer works as expected.

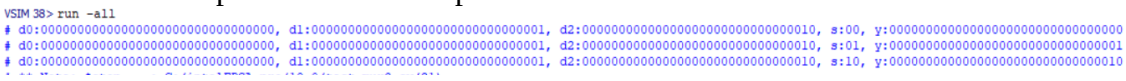


Figure 12, the simulation result of the 3-to-1 multiplexer

8. Main decoder

Op	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	ALUOp
0000011	1	000	1	0	01	00
0010011	1	000	1	0	00	10
0100011	0	001	1	1	00	00
0110011	1	xxx	0	0	00	10
1100011	0	010	0	0	00	01
1100111	1	011	1	0	10	00
1101111	1	011	1	0	10	00

Table 3, how the Main decoder works

In the main decoder, it outputs the control signal value depending on the opcode. In the testbench code, different opcodes are inputted to the main decoder and the control signals are shown in the simulation result. The table above matches simulation results. As a result, the main decoder works as expected.

```
# op:0000011 ,RegWrite:1 ,ImmSrc:000 ,ALUSrc:1, MemWrite:0, ResultSrc:01, ALUOp:00
# op:0010011 ,RegWrite:1 ,ImmSrc:000 ,ALUSrc:1, MemWrite:0, ResultSrc:00, ALUOp:10
# op:0100011 ,RegWrite:0 ,ImmSrc:001 ,ALUSrc:1, MemWrite:1, ResultSrc:00, ALUOp:00
# op:0110011 ,RegWrite:1 ,ImmSrc:xxx ,ALUSrc:0, MemWrite:0, ResultSrc:00, ALUOp:10
# op:1100011 ,RegWrite:0 ,ImmSrc:010 ,ALUSrc:0, MemWrite:0, ResultSrc:00, ALUOp:01
# op:1100111 ,RegWrite:1 ,ImmSrc:011 ,ALUSrc:1, MemWrite:0, ResultSrc:10, ALUOp:00
# op:1101111 ,RegWrite:1 ,ImmSrc:011 ,ALUSrc:1, MemWrite:0, ResultSrc:10, ALUOp:00
```

Figure 13, the simulation result of the main decoder

9. ALU decoder

   The ALU decoder receives information from the instructions and the ALUOp signal from the main decoder. A 3-bit ALUControl signal which specifies which operation ALU will perform is computed at the output. In the testbench code, different opcodes are inputted to the ALU decoder and the ALU control signals are shown in the simulation result. For example, in load instruction, the opcode is 0110011, funct3 is 111, ALUOp is 10, and funct7 has 1 in bit 5. The signal ALUControl should be 010 since ALU is used to add the register with the immediate number to find the address in the data memory, and this statement matched the simulation result below. As a result, the ALU decoder works as expected.

```
ModelSim> run -all
# op:0010011 ,funct3:000 ,funct7b5:x ,ALUOp:10 ,ALUControl:000
# op:0000011 ,funct3:010 ,funct7b5:x ,ALUOp:00 ,ALUControl:000
# op:1100011 ,funct3:000 ,funct7b5:x ,ALUOp:01 ,ALUControl:001
# op:0110011 ,funct3:000 ,funct7b5:1 ,ALUOp:10 ,ALUControl:001
# op:0110011 ,funct3:111 ,funct7b5:0 ,ALUOp:10 ,ALUControl:010
# op:0010011 ,funct3:101 ,funct7b5:1 ,ALUOp:10 ,ALUControl:111
```

Figure 14, the simulation result of the ALU decoder

The System Verilog code for each function unit is attached to the appendices.

4.5 System Verilog code realisation and simulation of both single-cycle and pipelined microprocessors

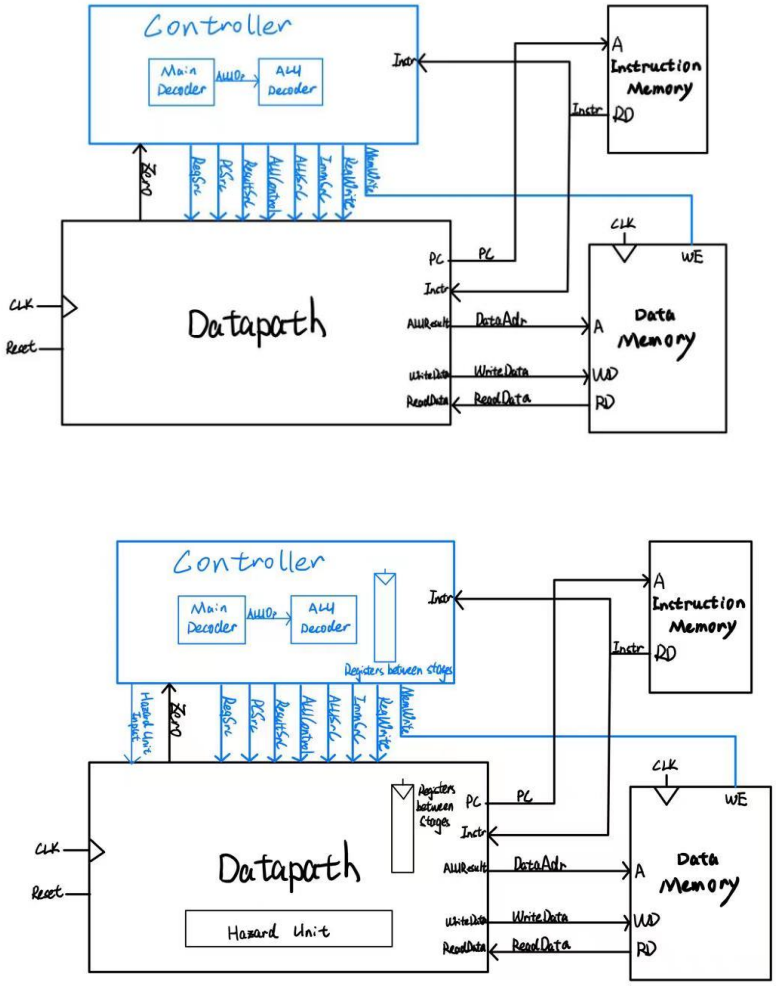


Figure 15, the layouts of a microprocessor in both microarchitectures

Figure 15 shows the general block diagrams for both single-cycle and pipelined RISC-V processors. The hierarchy of the design is also displayed in figure 16 from high to low.

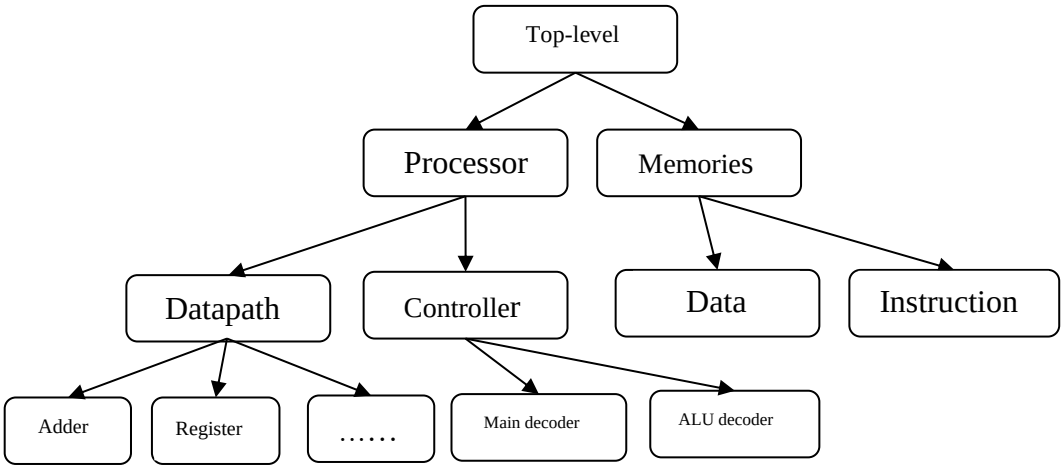


Figure 16, the hierarchy of the design

In the top-level module, the two microprocessors have the same number of input and output ports. Both of the designs have the clock and reset signals as the input. The outputs from them are the result from the ALU, the data memory address, and the ENABLE signal for the data memory.

In order to test the whole processor designs, a test program whose results in each step can be tracked to verify the processor is shown below.

PC address (in hexadecimal)	Instruction	Explanation (in decimal)
0	$addi\ x2, x0, 5$	$x2 = 5$
4	$addi\ x3, x0, 0xc$	$x3 = 12$
8	$addi\ x7, x3, -9$	$x7 = 3$
C	$or\ x4, x7, x2$	$x4 = 7$
10	$and\ x5, x3, x4$	$x5 = 4$
14	$add\ x5, x5, x4$	$x5 = 11$
18	$beq\ x5, x7, 0x30$	Branch is not taken since $x5 \neq x7$
1c	$addi\ x4, x0, 0$	$x4 = 0$
20	$beq\ x4, x0, 8$	Branch is taken to address 0x28 since $x4 = x0$
24	$addi\ x3, x0, 0$	Not execute
28	$addi\ x4, x4, 1$	$x4 = 1$
2c	$add\ x7, x4, x5$	$x7 = 12$
30	$sub\ x7, x7, x2$	$x7 = 7$
34	$sw\ x7, 0x54(x3)$	$memory[96] = 7$
38	$lw\ x2, 0x60(x0)$	$x2 = memory[96] = 7$
3c	$add\ x9, x2, x5$	$x9 = 18$
40	$jal\ x3, 8$	Jump to address 48, $x3 = 0x40 + 4 = 0x44$, which is 68 in decimal
44	$addi\ x2, x0, 1$	Not execute
48	$add\ x2, x2, x9$	$x2 = 25$
4c	$sw\ x2, 0x20(x3)$	$memory[100] = 25$
50	$beq\ x2, x2, 0$	A loop, always goes back to this instruction

Table 4, the test program

The testbench is also designed based on the test program. It checks whether the data stored in the memory is 25 with the address 100 at the end of the instruction since all previous instructions are relevant to the data and its memory. The “simulation succeed” will be reported if the data is 25 while the address is 100. Otherwise, if the save word instruction at program address 34 does not have the data 7 with address 96, there must be errors in the execution process of the previous instructions and the testbench will report “simulation failed”. The testbench code and the simulation results are shown below.

```
@author: Zhaoyu Wang
module testbench();
 logic clk;
 logic reset;
 logic [31:0] WriteData, DataAdr;
 logic MemWrite;
 // instantiate device to be tested
 top dut(clk, reset, WriteData, DataAdr, MemWrite);
 // initialize test
 initial
 begin
 reset <= 1; # 22; reset <= 0;
 end
 // generate clock to sequence tests
 always
 begin
 clk <= 1; # 5; clk <= 0; # 5;
 end
 // check results

initial begin
$monitor("clk:%b, reset:%b, memwrite:%b, ALUResult:%d, WriteData:%d",clk, reset, WriteData,
DataAdr, MemWrite);
$monitor("ALUResult:%d",DataAdr);
end

 always @(negedge clk)

 begin
 if(DataAdr == 100)begin//DataAdr is the alu result for the last instruction 100
$display("the data at the address is %b", WriteData);
end
 if(MemWrite) begin
 if(DataAdr === 100 & WriteData === 25) begin //100 25
 $display("Simulation succeeded");
 $stop;
 end else if (DataAdr !== 96)
begin
 $display("Simulation failed");
 $stop;
 end
 end
 end
endmodule
```

```
VSIM 9> run -all
# ALUResult:          5
# ALUResult:         12
# ALUResult:          3
# ALUResult:          7
# ALUResult:          4
# ALUResult:         11
# ALUResult:          8
# ALUResult:          0
# ALUResult:          1
# ALUResult:         12
# ALUResult:          7
# ALUResult:         96
# ALUResult:         96
# ALUResult:         18
# ALUResult:          8
# ALUResult:         25
# ALUResult:        100
# the data at the address is 00000000000000000000000000011001
# Simulation succeeded
# ** Note: $stop    : C:/intelFPGA_pro/18.0/testbench.sv(35)
```

Figure 17, the testbench and simulation result of the single-cycle processor

```
module testbench_pip();
 logic clk;
 logic reset;
 logic [31:0] WriteDataM;
logic [31:0] ALUResultM;
 logic MemWriteM;
logic [31:0]InstrF,PCF;

//logic [31:0]RD1E,r3;
 top_pip dut(clk, reset,WriteDataM,ALUResultM,MemWriteM,InstrF,PCF);
 // initialize test
 initial
 begin
 reset <= 1; # 22; reset <= 0;
 end
initial begin
$monitor("ALUResult is:",ALUResultM);
end
 // generate clock to sequence tests
 always
 begin
 clk <= 1; # 5; clk <= 0; # 5;
 end
 // check results
always @(negedge clk)

 begin
 if(ALUResultM == 100)begin//DataAdr is the alu result for the last instruction 100
$display("the data at the address is %b", WriteDataM);
end
 if(MemWriteM) begin
 if(ALUResultM === 100 & WriteDataM === 25) begin //100 25
 $display("Simulation succeeded");
 $stop;
 end else if (ALUResultM !== 96)
begin
 $display("Simulation failed");
 $stop;
 end
 end
 end
endmodule
```

```
# ALUResult is:         0
# ALUResult is:         x
# ALUResult is:         5
# ALUResult is:        12
# ALUResult is:         3
# ALUResult is:         7
# ALUResult is:         4
# ALUResult is:        11
# ALUResult is:         8
# ALUResult is:         0
# ALUResult is:         x
# ALUResult is:         1
# ALUResult is:        12
# ALUResult is:         7
# ALUResult is:        96
# ALUResult is:         0
# ALUResult is:        18
# ALUResult is:         8
# ALUResult is:         0
# ALUResult is:         x
# ALUResult is:        25
# ALUResult is:       100
# the data at the address is 00000000000000000000000000011001
# Simulation succeeded
```

Figure 18, the testbench and simulation result of the pipelined processor

There are 2 reasons for the occurrence of x in the simulation result of the pipelined microprocessor:

- Once the processor starts working, since the output of the ALU at the Memory stage is displayed, it takes 3 clock cycles to propagate the first instruction from the Fetch stage to the Memory stage. The output will be x if the tow inputs of the ALU in the Execute stage is not specified by the instructions. As a result, x will be computed by the ALU and passed to the Memory stage before the first instruction reaches this stage.
- The B-type and J-type instructions will also lead the ALU output to be zero since the processor predicts that no branches are taken when dealing with these two types of instructions and flushes the two following instructions when the branch is decided to be taken in the Execute stage. Thus, the ALU inputs are not specified again before the target instruction of the B-type and J-type instructions reaches the Execute stage and Memory stage.

4.6 Implementation of the two designs at the FPGA to execute a high-level language program

In order to verify the behaviour of the designs in handling real-world problems, the processors are updated to the FPGA board to experimentally test the design. The processors then execute a series of instructions to calculate the first 7 numbers in the Fibonacci series. Each number in the Fibonacci series is the sum of the previous two numbers [17]. By definition, it is very easy to write a high-level-language based program, for example, in Python.

```python
#initiallize save variable used
n1 = 0
n2 = 1
count = 2
number = 7#how many numbers to be found in the series

print(f"nth is {n1}")
print(f"nth is {n2}")
#the loop will print out the fibonaci
while count!=number:#for number of terms greater than 1
    nth = n1+n2 #the next number in fibonacci series
    print(f"nth is {nth}")
    n1 = n2 #update the two operating varibale
    n2 = nth
    count += 1 #counter plus 1
```

Figure 19, a python program to compute the first 7 numbers in the Fibonacci series

However, this program is far from the one that the processor could understand and execute. As we introduced, the first step to handle this program is converting it into the assembly language description. First of all, the registers in the processors are assigned to each of the variables in the program in figure 19. In RV32I, the 32 registers are listed below:

registers	description
$x0/zero$	Hardwired zero
$x1/ra$	Return address
$x2/sp$	Stack pointer
$x3/gp$	Global pointer
$x4/tp$	Thread pointer
$x5-x7/t0-t2$	Temporary
$x8/s0\ /fp$	Saved register, frame pointer
$x9/s1$	Saved register
$x10-x11/a0-a1$	Function argument, return value
$x12-x17/a2-a7$	Function argument
$x18-x27/s2-s11$	Saved register
$x28-x31/t3-t6$	temporary

Table 5, the RV32I registers [6]

In the program above, the variable “n1” is assigned to x9, the variable “n2” is assigned to x18, “count” is assigned to x19, “number” is assigned to x20 and “nth” is assigned to x5. The assembly for the code in figure 20 is shown below:

```
# x9=n1 x18=n2 x19=count x20=number x5=nth

#addi s1,zero,0
#addi s2,zero,1
#addi s3,zero,2
#addi s4,zero,7
#beq s3,s4,24 if branch is taken, the loop condition is not met,move to sw instruction
#add t0,s1,s2
#addi s1,s2,0
#addi s2,t0,0
#addi s3,s3,1
#jal x0,20  jump back to the branch instruction to keep looping
#sw s2,(4)x0
```

Figure 20, the assembly code for the Fibonacci program

One more step which translates the assembly language into machine language interpreted merely by 1 and 0 is taken since the processor can only understand the 1-and-0 representation. By importing an open-source package, RISC-V assembler, the machine codes for all types of instructions can be computed from the assembly language [16]. The 32-bit instructions are displayed in figure 21.

Figure 21, the machine codes for the Fibonacci program

Those machine codes are the ones the machine is able to tackle and are written into the instruction memory. The Quartus Prime Lite software synthesised the circuit design and uploaded the design to the FPGA board. In order to demonstrate how processors work, the output from the ALU is displayed by the peripherals on the FPGA board. In this project, the lower 8 bits from the result of the ALU are linked to LED by the pin planner. The bit is 1 if the LED at the corresponding position is on and vice versa. For example, if the result of the ALU is 5, which is 00000101 in binary form, only the first and the third LED will be on.

Table 3-1 Pin Assignments for Push-buttons

Signal Name	*FPGA Pin No.*	*Description*	*I/O Standard*
KEY[0]	PIN_J15	Push-button[0]	3.3V
KEY[1]	PIN_E1	Push-button[1]	3.3V

Table 3-2 Pin Assignments for LEDs

Signal Name	*FPGA Pin No.*	*Description*	*I/O Standard*
LED[0]	PIN_A15	LED Green[0]	3.3V
LED[1]	PIN_A13	LED Green[1]	3.3V
LED[2]	PIN_B13	LED Green[2]	3.3V
LED[3]	PIN_A11	LED Green[3]	3.3V
LED[4]	PIN_D1	LED Green[4]	3.3V
LED[5]	PIN_F3	LED Green[5]	3.3V
LED[6]	PIN_B1	LED Green[6]	3.3V
LED[7]	PIN_L3	LED Green[7]	3.3V

Figure 22, the pin assignments from the user manual [17]

The clock and reset signals are also linked to pin J15 and E1 respectively. As the pushbuttons are pressed, 1 is asserted at the corresponding signal. The signal will be assigned 0 if the corresponding pushbutton is released.

Node Name	Direction	Location
ALUResultM_8[7]	Output	PIN_L3
ALUResultM_8[6]	Output	PIN_B1
ALUResultM_8[5]	Output	PIN_F3
ALUResultM_8[4]	Output	PIN_D1
ALUResultM_8[3]	Output	PIN_A11
ALUResultM_8[2]	Output	PIN_B13
ALUResultM_8[1]	Output	PIN_A13
ALUResultM_8[0]	Output	PIN_A15
clk	Input	PIN_J15
reset	Input	PIN_E1

Figure 23, the pin configuration in the experiment

After updating the design to the FPGA board, the pushbutton E1 is pressed to reset the processor. The button J15 is then pressed and released to provide a complete clock cycle. In both of processor designs, the result of one instruction is displayed at the LED by pushing the button once, and the next instruction is executed by the processor at the next press. From the LEDs, the processors are shown to work correctly.

nth clock cycle	$LED_{7:0}$ result in binary form	description
0	00000000	$n1 = 0$
1	00000001	$n2 = 1$
2	00000010	$count = 2$
3	00000111	$number = 7$
4	11111011	Loop condition met, $count \neq number$ the loop continues
5	00000001	$nth = n1 + n2 = 1$
6	00000001	$n1 = n2 = 1$
7	00000001	$n2 = nth = 1$
8	00000011	$count = count + 1 = 3$
9	11101100	$jal\ x0, -20$, and jump back to the beq instruction. Loop goes on
10	11111100	Loop condition met, $3 \neq 7$, the loop continues
11	00000010	$nth = n1 + n2 = 2$
12	00000001	$n1 = n2 = 1$
13	00000010	$n2 = nth = 2$
14	00000100	$count = count + 1 = 4$
15	11101100	$jal\ x0, -20$, and jump back to the beq instruction. Loop goes on
16	11111101	Loop condition met, $4 \neq 7$, the loop continues
17	00000011	$nth = n1 + n2 = 3$
18	00000010	$n1 = n2 = 2$
19	00000011	$n2 = nth = 3$

20	00000101	$count = count + 1 = 5$
21	11101100	$jal\ x0, -20$, and jump back to the beq instruction. Loop goes on
22	11111110	Loop condition met, $5 \neq 7$, the loop continues
23	00000101	$nth = n1 + n2 = 5$
24	00000011	$n1 = n2 = 3$
25	00000101	$n2 = nth = 5$
26	00000110	$count = count + 1 = 6$
27	11101100	$jal\ x0, -20$, and jump back to the beq instruction. Loop goes on
28	11111111	Loop condition met, $6 \neq 7$, the loop continues
29	00001000	$nth = n1 + n2 = 8$
30	00000101	$n1 = n2 = 5$
31	00001000	$n2 = nth = 8$
32	00000111	$count = count + 1 = 7$
33	11101100	$jal\ x0, -20$, and jump back to the beq instruction. Loop goes on
34	00000000	Condition to quit the loop is met, $7 = 7$, the loop breaks
35	00000100	Save the data at address 0+4= 4
36	00000000	No more instructions, program finishes

Table 6, the experiment result and explanation of single-cycle processor

nth clock cycle	$LED_{7:0}$ result (ALUResultM) in binary form	Description (Result of the ALU in ***Memory*** stage)
0	00000000	
1	00000000	
2	00000000	
3	00000000	$n1 = 0$
4	00000001	$n2 = 1$
5	00000010	$count = 2$
6	00000111	$number = 7$
7	11111011	Loop condition met, $2 \neq 7$, the loop continues
8	00000001	$nth = n1 + n2 = 1$
9	00000001	$n1 = n2 = 1$
10	00000001	$n2 = nth = 1$
11	00000011	$count = count + 1 = 3$
12	11101100	$jal\ x0, -20$, and jump back to the beq instruction. Loop goes on

13	00000000	Hazard unit works at jump and link instruction, flush is taken and ALU receives 0 at both inputs
14	00000000	Hazard unit works, flush is taken and ALU receives 0 at both inputs. The next instruction will jump back to the beq instruction.
15	11111100	Loop condition met, $3 \neq 7$, the loop continues
16	00000010	$nth = n1 + n2 = 2$
17	00000001	$n1 = n2 = 1$
18	00000010	$n2 = nth = 2$
19	00000100	$count = count + 1 = 4$
20	11101100	$jal\ x0, -20$, and jump back to the beq instruction. Loop goes on
21	00000000	Hazard unit works at jump and link instruction, flush is taken and ALU receives 0 at both inputs
22	00000000	Hazard unit works, flush is taken and ALU receives 0 at both inputs. The next instruction will jump back to the beq instruction.
23	11111101	Loop condition met, $4 \neq 7$, the loop continues
24	00000011	$nth = n1 + n2 = 3$
25	00000010	$n1 = n2 = 2$
26	00000011	$n2 = nth = 3$
27	00000101	$count = count + 1 = 5$
28	11101100	$jal\ x0, -20$, and jump back to the beq instruction. Loop goes on
29	00000000	Hazard unit works at jump and link instruction, flush is taken and ALU receives 0 at both inputs
30	00000000	Hazard unit works, flush is taken and ALU receives 0 at both inputs. The next instruction will jump back to the beq instruction.
31	11111110	Loop condition met, $5 \neq 7$, the loop continues
32	00000101	$nth = n1 + n2 = 5$
33	00000011	$n1 = n2 = 3$

34	00000101	$n2 = nth = 5$
35	00000110	$count = count + 1 = 6$
36	11101100	$jal\ x0, -20$, and jump back to the beq instruction. Loop goes on
37	00000000	Hazard unit works at jump and link instruction, flush is taken and ALU receives 0 at both inputs
38	00000000	Hazard unit works, flush is taken and ALU receives 0 at both inputs. The next instruction will jump back to the beq instruction.
39	11111111	Loop condition met, $6 \neq 7$, the loop continues
40	00001000	$nth = n1 + n2 = 8$
41	00000101	$n1 = n2 = 5$
42	00001000	$n2 = nth = 8$
43	00000111	$count = count + 1 = 7$
44	11101100	$jal\ x0, -20$, and jump back to the beq instruction. Loop goes on
45	00000000	Hazard unit works at jump and link instruction, flush is taken and ALU receives 0 at both inputs
46	00000000	Hazard unit works, flush is taken and ALU receives 0 at both inputs. The next instruction will jump back to the beq instruction.
47	00000000	Condition to quit the loop is met, $7 = 7$, the loop breaks
48	00000000	Hazard unit works at jump and link instruction, flush is taken and ALU receives 0 at both inputs
49	00000000	Hazard unit works, flush is taken and ALU receives 0 at both inputs. The next instruction will jump back to the beq instruction.
50	00000100	Save the data at address 0+4= 4
51	00000000	No more instructions, program finishes

Table 7, the experiment result and explanation of single-cycle processor

From the implementation results, the expected output is received at each step. This shows that the processors are working correctly.

4.7 Power-saving technique: Clock gating

The most significant source in the dynamic power consumption is the switching at the capacitance, where the internal capacitor charges and discharges itself. This is due to the characteristics of the CMOS technology of the transistors. However, to minimise the dynamic power, the proposed idea, clock gating is applied to reduces the unnecessary switching by blocking the clock signal that drives the clock-edge-sensitive function blocks that are idle at that clock cycle.

In order to gate the clock, the gating element should receive 2 inputs: the ENABLE signal which decided if the function block is working, and the clock signal. When the ENABLE signal is asserted, the function block is working at the edges of the clock and the gated clock is transparent to the original clock signal. Otherwise, the gated clock which drives the building block will be 0 and prevent the unnecessary switching.

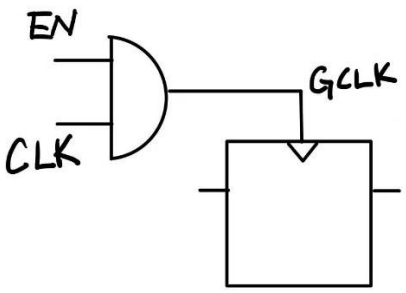


Figure 24, the simplest clock gating

However, the glitches in the gated clock signal could be induced if the ENABLE signal is not under control. For example, the figure below shows one of the situations where the glitched are introduced. The glitches are very dangerous to the time sequence and can lead to errors.

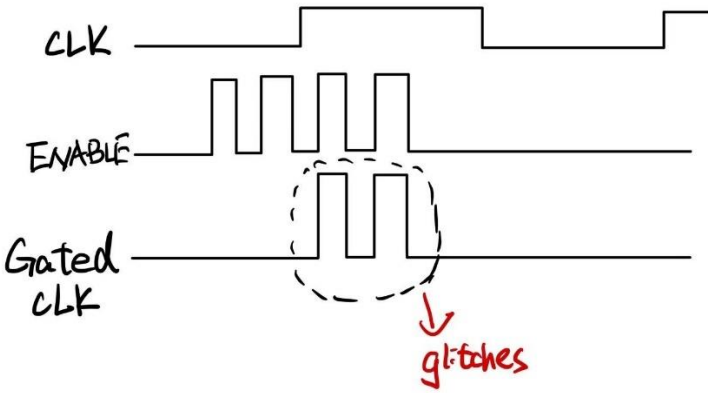


Figure 25, the formation of glitches

In order to solve the problem from glitches, the latch is added to the gating design in order to introduce a delay. The design layout is displayed in figure 26:

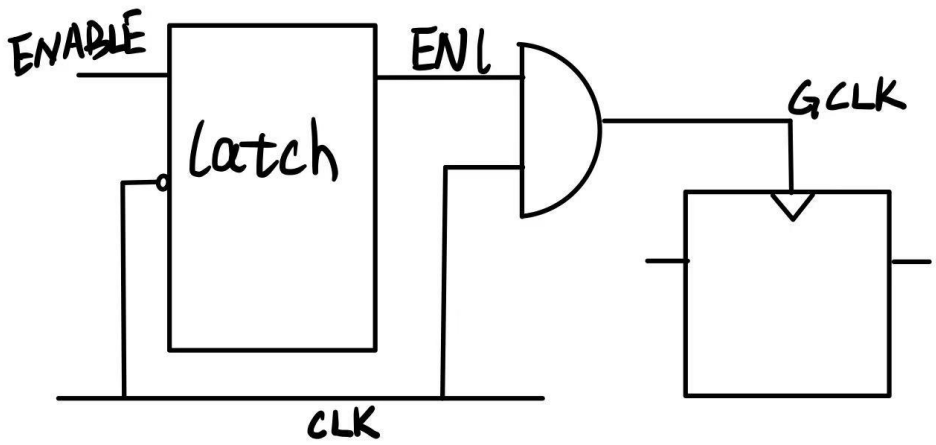


Figure 26, modified clock gating

The latch is only transparent when the clock is low. As a result, the glitch of the ENABLE signal only appears when clock is zero. Since the new ENABLE signal, ENl and the clock which is low at this time are inputs of the AND gate, the glitches can be cancelled. This approach can significantly improve the performance of the clock gating.

After applying the clock gating to the clock-sensitive elements such as register files and data memories in both single-cycle and pipelined microprocessors, the same simulation process as the one described in 4.5 is carried out. The results from both processors show the clock gating produces the correct clock signals and the design is working correctly with the clock gating unit.

4.8 Performance and power analysis

Performance and power consumption are two important factors in choosing the most suitable processor. Since the single-cycle and pipelined microprocessors are different in both structures and the ways to deal with instruction (single-cycle processor executes one instruction per cycle and the pipelined processor deals with 5 instructions in different stages in one cycle), the difference in their power consumption and performance can be compared.

4.8.1 Performance analysis

This section analyses the performance of the processors from two parameters: the instruction-per-cycle (IPC)/ cycle-per-instruction (CPI) and the clock period.

4.8.1.1 The Instruction-per-cycle (IPC) or Cycle-per-instruction (CPI)

As the name suggests, CPI represents the number of clock cycles taken to finish one specific instruction. The reciprocal of the CPI, the IPC, can directly measure the throughput of the processor in one clock cycle. To analyse and compare the CPI between two microarchitecture designs, the result from the experimental result in section 4.6 is used to provide significant proof for their performance.

4.8.1.1.1 Single-cycle microprocessor

In the single-cycle microprocessor, all necessary steps are taken in 1 complete clock cycle. The instruction with the address PC is fetched from the instruction memory and information

from the instruction is passed to both the datapath and controller. The data is logically or arithmetically manipulated, and the result is written back to the memory or the register file at the edge of this clock cycle. From the experimental result in section 4.6, the result of ALU in one instruction is displayed at each clock cycle. As a result, the single-cycle processor is able to deal with 1 instruction in each clock cycle, and the CPI is therefore, 1.

4.8.1.1.2 Pipelined Microprocessor

Similarly, since one new instruction is fetched from the memory in each cycle, the pipelined processor should ideally have 1 in CPI. However, from the experimental result, the pipelined processor takes 15 more clock cycles to finish the program, and this results in an CPI which is greater than 1. There are two reasons for the greater CPI of pipelined design compared with the single-cycle processor. First of all, since the output from the pipelined processor is introduced from the memory stage, it takes 3 clock cycles for the first instruction to reach these stages, and this introduces 3 more clock cycles. Second, the J-type and B-type instructions introduces control hazards and flushes are applied to the processor to solve the hazards. The processor predicts that the branches are not taken when handling the J-type and B-type instructions, continues executing the following instructions, and decides to flush the two instructions which follows the branch if the condition of taking the branch is satisfied. The wasted instruction cycles are termed branch misprediction penalty [6], and this penalty is determined by the product of the fraction of mispredicted branches, and the number of lost execution cycles per mispredicted branch [18]. For example, in clock cycle 48 and 49 from table x, the processor is supposed to execute the save word instruction after the loop is broken and program should be finished. However, the ALU result in memory stage displays zero in clock 48 and 49. The reason is that the pipelined processor detects control hazard at the B-type instruction, $beq\ s3, s4, 24$. Since the branch is taken at the last round of the loop and the processor predicts that the branch is not taken at the beginning, the two following instructions are flushed, and zeros are delivered to the ALU input in cycle 48 and 49. The two redundant instructions result in the branch misprediction penalty and increases the CPI. Thus, the CPI of the pipelined design should be greater than 1, depending on how much the stalls and flushed are taken when dealing with the hazards.

4.8.1.1.3 CPI Summary

The average CPI of both processor designs can be calculated when the processors execute the Fibonacci program. Ideally 36 instructions should be executed since the loop repeats handling a number of instructions. The single-cycle processor will have an IPC of 1. In the pipelined processor, there are 24 (66.67%) R or I-Type ALU instructions (add and add immediate), 1 (2.78%) save word instruction, 6 (16.67%) branches and 5 (13.89%) jumps. In addition, 1 of the 6 branches (16.67%) are taken and mispredicted so that 3 cycles are taken to execute this branch. All of the 5 jumps take 3 cycles, while the R- and I-Type ALU instructions as well as the save word instruction takes 1 cycle to complete. There are no load word instructions, so there is no risk of stalls in the processor.

The branch in this task has a CPI of:

$$16.67\% \times 3 + 83.33\% \times 1 = 1.33$$

Hence, the average CPI for the pipelined microprocessor is:

$$0.667 + 0.028 + 0.167 \times 1.33 + 0.139 \times 3 = 1.334$$

However, from the calculation process above, the CPI for a processor might not be fixed. It is fixed for the single-cycle processor, i.e., the CPI is 1 for single-cycle processor, but the cycle-per-instruction varies in the pipelined processor, depending on what the program includes and how much stalls and flushes the processor will perform. In summary, the CPI of the pipelined processor will be greater, or ideally equal to 1. The CPI of the pipelined processor is always greater than or equal to the CPI in the single-cycle processor, disregarding the length of cycle period.

4.8.1.2 Clock period

Using merely the CPI to demonstrate the performance of the processor is not enough, since the CPI only consider how many clocks cycles the processor takes but not the length of the cycle period. The clock periods are also different in the two processors and depend on the critical path of the microarchitecture.

With the advance of the CMOS technology, the delay of the circuit elements is becoming lower and lower. In the estimation of clock period, the assumption, that the processor is built in a 7-nm CMOS manufacturing process is applied. From the existed data, the delays of the circuit elements are shown in the table below [6]:

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR/inverter gate	t_{inv}	10
ALU	t_{ALU}	120
Control Unit	$t_{control}$	25
Extend Unit	t_{ext}	35
Memory Read	t_{mem}	200
Register file Read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Table 8, the delays of the circuit elements with 7-nm CMOS manufacturing process

From the table, the execution of ALU, memory and register file requires longer time than the others.

The clock period in the single-cycle microprocessor is determined by the instruction with the longest critical path. The load word instruction takes the longest time to execute among the 21 selected instructions since the processor reads both the instruction memory, data memory and register file, and uses ALU to perform calculation, i.e., all of the elements which takes longer time are used in this instruction. The critical path is labelled in figure 27.

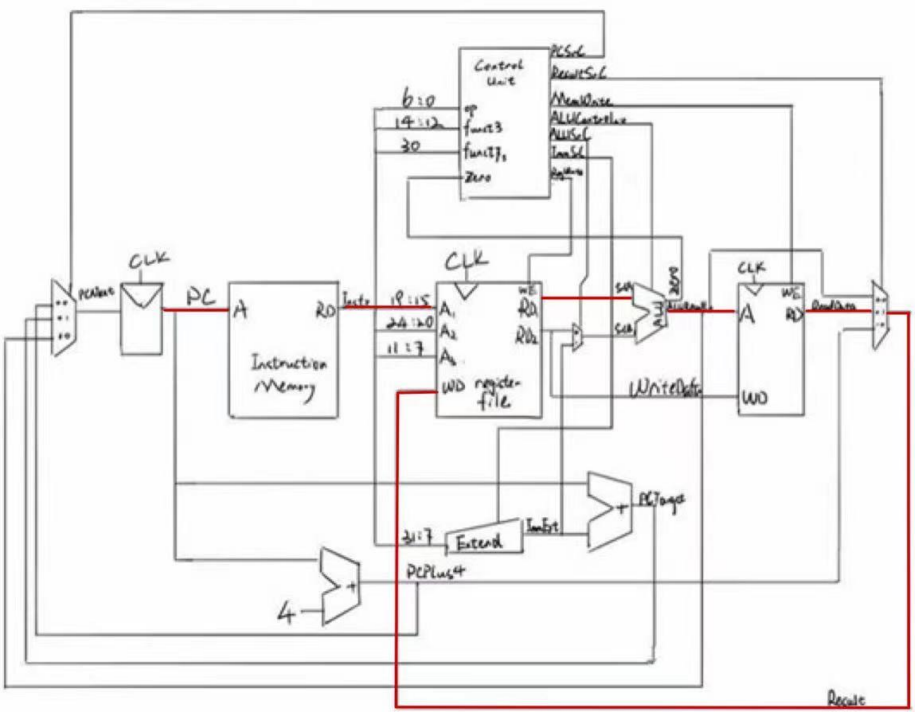


Figure 27, the critical path of the single-cycle processor

From the table above, the time to finish the critical path for the load word instruction is:

$$T_{clk\_sing} = 40 + 200 + 100 + 60 + 120 + 200 + 30 = 750ps$$

As a result, each clock cycle for the single-cycle processor is 750ps.

In the pipelined microprocessor, since the processor is split into 5 stages and these stages works simultaneously, the clock cycle depends on the critical path in each stage. Based on what each stage does and which elements each stage will use, the stage which takes the longest time to finish its work can be figured out. To make the processor synchronous, the clock cycle is defined from the stage whose executing time is the longest. The equation below is derived to find the clock cycle.

$$T_{clk\_pip} = max\begin{cases} t_{pcq} + t_{setup} + t_{mem} \\ 2 \times (t_{RFread} + t_{setup}) \\ t_{pcq} + t_{setup} + 7 \times t_{mux} + t_{inv} + t_{ALU} \\ t_{pcq} + t_{setup} + t_{mem} \\ 2 \times (t_{mux} + t_{pcq} + t_{RFsetup}) \end{cases}$$

$$T_{clk\_pip} = max\begin{cases} 40 + 50 + 200 = 290ps \\ 2 \times (50 + 100) = 300ps \\ 40 + 50 + 7 \times 30 + 10 + 120 = 430ps \\ 40 + 50 + 200 = 290ps \\ 2 \times (30 + 40 + 60) = 260ps \end{cases}$$

From the result above, the Execute stage could take the longest time, when there is data hazard and the ALU input comes from the destination register of the Writeback stage. As the result, the minimum clock period for the pipelined processor design is 430ps.

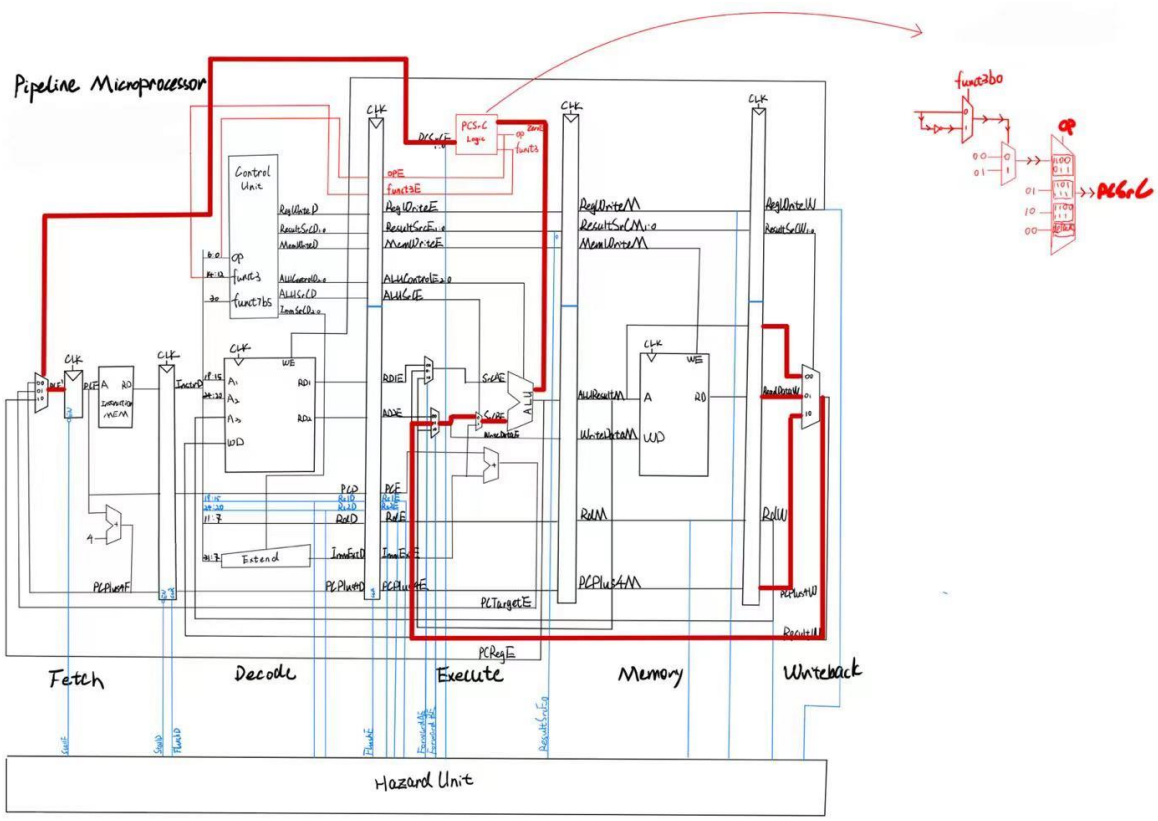


Figure 28, the critical path in the Execute stage

To conclude, the single-cycle microprocessor has 750ps clock period, while the pipelined processor has 430ps clock period. In comparison, the pipelined processor could save 42.7% clock cycles. To finish executing the same instructions 21930 ps, while the single-cycle processor takes 27000ps.

4.8.2 Power analysis

In order to reduce the power consumption, unnecessary instructions are excluded from the instruction set, and clock gating is applied to the processor to reduce dynamic power consumption.

Comparing the RV32I instruction set with the instruction set in this project, most of the excluded instructions are R-type and I-type, which use ALU to perform calculations. The removal of those unnecessary reduces the categories of ALU calculation, and therefore reduce the number of gates in the combinational logic of ALU, and the bits of the control signal ALUControl. For example, the R-Type instruction “set less than” which stores the smaller source register to the destination register. However, this instruction can be replaced by writing the assembly code in another way. As a result, the ALU does not require to include an additional multiplexer and, a comparator and one bit more control signal and saves the power consumption since the number of gates and the static power is reduced.

The clock gating is also an effective approach to minimise dynamic power consumption. It works by reducing unnecessary clock switches applied to the clock-sensitive function blocks when they are idle. Since the power consumption is determined by the equation $p = \alpha C V^2 f$

from equation 1.2, the clock gating reduces the dynamic power by minimising the α term, the number of switching activities.

The rate of switching activities depends on the instructions the processor executes, since not all of the building blocks in the microarchitectures uses the clock-sensitive elements. Table 9 below concludes the elements that could benefit from the clock gating in each type of the instructions.

Instruction Type	Clock-edge-sensitive elements which are idle and can save power using clock gating
R-Type	The data memory writing mode
I-Type	The data memory writing mode
S-Type	No elements are idle
B-Type	The data memory writing mode and the register file writing mode
J-Type	The data memory writing mode

Table 9, the idle clock-edge-sensitive elements in each type of instructions

The assumption should be made when calculating the percentage of dynamic power saved using clock gating. Since both the register file and the data memory are made up by SRAMs, we assume that the power consumption for them is the same. From the implementation part in section 4.6, ideally 36 instructions should be executed. There are 24 R or I-Type ALU instructions (add and add immediate), 1 save word instruction, 6 branches and 5 jumps. Without the clock gating, there are 72 clock switches at both the data memory and register file respectively (clock switches from 1 to 0 and 0 to 1), and in total, 144 switches. However, by applying the clocks gating, the number of clock switches becomes:

$$(24 + 1 \times 2 + 5) \times 2 = 124$$

This represents a saving of 13.9% in dynamic power consumption.

5 Conclusion and future work

In this project, two microarchitectures are proposed and designed based on the selected instruction set from the RISC-V standard which can reduce the design complexity. 21 instructions which are used most commonly are selected to design the microarchitectures. Both the single-cycle and pipelined processors are designed and simulated using Modelsim. The designs are also synthesised and tested using the field-programmable gate array development board. From the experimental result, the pipelined processor is 18.7% faster than the single-cycle processor (since the time taken to finish the same instruction in the pipelined processor is 18.7% shorter). The clock gating is a powerful technique to minimise the dynamic power can save ideally 13.9% dynamic power in the experiment when executing specific instructions. In conclusion, two main goals in this project are satisfied: complexity is reduced by selecting necessary instructions from the whole instruction set, while the clock gating minimises the power consumption.

In the future, the CPI of the design can be improved by applying more advanced microarchitectures such as superscalar designs. More instructions can be included in the instruction set so that the processor could deal with more tasks. In addition, other power-

saving techniques such as frequency switching could also be included in this design to reduce the unnecessary power dissipation as much as possible.

6 Reference

[1]: http://www.hubspire.com/what-is-a-cpu-and-what-is-its-function/ “Function of a Microprocessor”. Hubspire. 2017-04-26. Last accessed 2022 March 21

[2]: Adegbija T, Rogacs A, Patel C, et al. Microprocessor optimizations for the internet of things: A survey[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2017, 37(1): 7-20.

[3]: Berezinski, John. "RISC: Reduced Instruction set Computer". Department of Computer Science, Northern Illinois University. Archived from the original on 28 February 2017.

[4]: RISC-V International (riscv.org) RISC-V official site, last accessed 2022 March 25

[5]: Patterson D, Waterman A. The RISC-V Reader: an open architecture Atlas[M].

Strawberry Canyon, 2017

[6]: Harris S, Harris D. Digital Design and Computer Architecture: RISC-V Edition[M].

Morgan Kaufmann, 2021.

[7]: Hamann HF, et al. “Hotspot-limited microprocessors: Direct temperature and power

distribution measurements.” Solid-State Circuits, IEEE Journal of. 2007 Jan;42(1):56-65

[8]: Venkatachalam V, Franz M. Power reduction techniques for microprocessor systems[J]. ACM Computing Surveys (CSUR), 2005, 37(3): 195-237.

[9]: Nafkha A, Palicot J, Leray P and Louët Y. “Leakage power consumption in FPGAs: thermal analysis.” InWireless Communication Systems (ISWCS), 2012 International Symposium on 2012 Aug 28 (pp. 606-610). IEEE.

[10]: El-Razek M A, Abdelhalim M B, Issa H H. Dynamic power reduction of microprocessors for IoT applications[J]. Journal of Advanced Vehicle System, 2016, 2(1): 10-20.

[11]: Sherwood W. Simulation hierarchy for microprocessor design[C]//Proceedings of the Symposium on Design Automation and Microprocessors. 1977: 44-49.

[12]: Sinkov A. Programming the IBM 1401: A Self-Instructional Programmed Manual[J]. 1963.

[13]: Muchnick S. Advanced compiler design implementation[M]. Morgan kaufmann, 1997.

[14]: Waterman A, Lee Y, Patterson D, et al. The RISC-V instruction set manual[J]. Volume I: User-Level ISA’, version, 2014, 2.

[15]:https://github.com/andrescv/Jupiter#:~:text=Jupiter%20is%20an%20open%20source%20and%20education-oriented%20RISC-V,described%20in%20the%20user-level%20instruction%20set%20manual%201. Last accessed 2022 March 21

[16]: https://byjus.com/maths/fibonacci-numbers/ “What is Fibonacci Number”, last accessed 2022 March 25

[17]: "DEO-Nano User Manual", User Manual Version 1.8. Terasic Technologies Inc, 2012.

[18]: S. Eyerman, J. E. Smith and L. Eeckhout, "Characterizing the branch misprediction penalty," 2006 IEEE International Symposium on Performance Analysis of Systems and Software, 2006, pp. 48-58, doi: 10.1109/ISPASS.2006.1620789.

7 Appendices

7.1 System Verilog code

```
//@author: Zhaoyu Wang zceezw1@ucl.ac.uk  single cycle
module top(input logic clk,reset,
                output logic [31:0]WriteData,DataAdr,
                output logic  MemWrite);
    logic [31:0] PC, Instr, ReadData;
     // instantiate processor and memories
    riscvsingle rvsingle(clk, reset, PC, Instr, MemWrite,
    DataAdr, WriteData, ReadData);
    instru_mem imem(PC, Instr);
    data_mem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);

endmodule

//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
module riscvsingle(input logic clk, reset,
                            output logic [31:0] PC,
                            input logic [31:0] Instr,
                            output logic MemWrite,
                            output logic [31:0] ALUResult, WriteData,
                            input logic [31:0] ReadData);
    //logic ALUSrc, RegWrite, Jump, Zero;
    logic ALUSrc, RegWrite,Zero;

    logic [1:0] ResultSrc,PCSrc;
    logic [2:0] ImmSrc,ALUControl;
    //controller c(Instr[6:0], Instr[14:12], Instr[30], Zero,ResultSrc, MemWrite, PCSrc,
                //ALUSrc, RegWrite, Jump,ImmSrc, ALUControl);
    controller c(Instr[6:0], Instr[14:12], Instr[30], Zero,ResultSrc, MemWrite, PCSrc,
                ALUSrc, RegWrite, ImmSrc, ALUControl);

    datapath dp(clk, reset, ResultSrc, PCSrc,ALUSrc, RegWrite,ImmSrc, ALUControl,
                Zero, PC, Instr,ALUResult, WriteData, ReadData);
endmodule
```

```
//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
module controller(input logic [6:0]op,
                  input logic [2:0]funct3,
                  input logic funct7b5,
                  input logic zero,
                  output logic [1:0] ResultSrc,
                  output logic MemWrite,
                  output logic [1:0]PCSrc,
                  output logic ALUSrc,
                  //output logic RegWrite, Jump,
                  output logic RegWrite,
                  output logic [2:0]ImmSrc,
                  output logic [2:0]ALUControl);
    logic [1:0]ALUOp;
    logic btypezero; //????beq/bne?zero??

    //main_decoder
md(op,RegWrite,ImmSrc,ALUSrc,MemWrite,ResultSrc,Branch,ALUOp,Jump);
    main_decoder md(op,RegWrite,ImmSrc,ALUSrc,MemWrite,ResultSrc,ALUOp);

    alu_decoder ad(op,funct3,funct7b5,ALUOp,ALUControl);

    mux2_2 m2(zero,~zero,funct3[0],btypezero);//according to funct3[0],if beq,sub result
directly shows if registers are equal
//if bne, zero is the opposite situation of result of sub---zero=0,bne's zero is 1
    always_comb
    begin
        case(op)
            7'b1101111: PCSrc = 2'b01;//jal: choose 01
            7'b1100011: PCSrc = btypezero?2'b01:2'b00;
            7'b1100111: PCSrc = 2'b10;//jalr,choose 10:alu reault
            default: PCSrc = 2'b00;
        endcase
    end
    //assign PCSrc = Branch&btypezero|Jump;

endmodule
```

```
//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
module main_decoder(input logic [6:0]op,
                    output logic RegWrite,
                    output logic [2:0]ImmSrc,
                    output logic ALUSrc,
                    output logic MemWrite,
                    output logic [1:0]ResultSrc,
                    output logic [1:0]ALUOp

                    );
    logic [9:0]control;

    //assign {RegWrite,ImmSrc,ALUSrc,MemWrite,ResultSrc,Branch,ALUOp,Jump}=control;
    assign {RegWrite,ImmSrc,ALUSrc,MemWrite,ResultSrc,ALUOp}=control;

    always_comb
        case(op)
            /*
            7'b0000011: control = 12'b1_000_1_0_01_0_00_0;//3
            7'b0010011:  control = 12'b1_000_1_0_00_0_10_0;//19
            7'b0100011:  control = 12'b0_001_1_1_00_0_00_0;//35
            7'b0110011:  control = 12'b1_xxx_0_0_00_0_10_0;//51
            7'b1100011:  control = 12'b0_010_0_0_00_1_01_0;//99
            7'b1100111:  control = 12'b1_011_1_0_10_0_00_1;//103 modify alusrc of jalr
from 0 to 1
            7'b1101111: control = 12'b1_011_1_0_10_0_00_1;//111 jal dont care alusrc
vakue and aluop
            default:  control = 12'bx_xxx_x_x_xx_x_xx_x;//other cases
            */
            7'b0000011: control = 10'b1_000_1_0_01_00;//3
            7'b0010011:  control = 10'b1_000_1_0_00_10;//19
            7'b0100011:  control = 10'b0_001_1_1_00_00;//35
            7'b0110011:  control = 10'b1_xxx_0_0_00_10;//51
            7'b1100011:  control = 10'b0_010_0_0_00_01;//99
            7'b1100111:  control = 10'b1_011_1_0_10_00;//103 modify alusrc of jalr from 0
to 1
            7'b1101111: control = 10'b1_011_1_0_10_00;//111 jal dont care alusrc vakue
and aluop
            default:  control = 10'bx_xxx_x_x_xx_xx;//other cases
        endcase



endmodule
```

```
//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
module alu_decoder(input logic [6:0]op,
                   input logic     [2:0]funct3,
                   input logic funct7b5,
                   input logic [1:0]ALUOp,
                   output logic [2:0]ALUControl);
    logic RtyprSub;

    assign RtypeSub = funct7b5&op[5];

    always_comb
        case(ALUOp)
            2'b00:    ALUControl = 3'b000;//lw sw jalr
            2'b01:    ALUControl = 3'b001;//subtraction for branch
            default: case(funct3)
                        3'b000:   if(RtypeSub)
                                      ALUControl = 3'b001;//sub,subi
                                  else
                                      ALUControl = 3'b000;//add/addi
                        3'b001:   ALUControl = 3'b101;
                        3'b100: ALUControl = 3'b100;
                        3'b101: if(funct7b5)
                                      ALUControl = 3'b111;
                                  else
                                      ALUControl = 3'b110;
                        3'b110:   ALUControl = 3'b011;
                        3'b111: ALUControl = 3'b010;
                        default: ALUControl = 3'bxxx;
                     endcase
        endcase


endmodule
```

```
//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
module datapath_pip(input logic clk,reset,//pipeline datapath
                    input logic [1:0]PCSrcE,
                    input logic[31:0]InstrF,
                    input logic [2:0] ImmSrcD,
                    input logic ResultSrcE0,
                    input logic [2:0]ALUControlE,
                    input logic ALUSrcE,RegWriteM,
                    input logic [31:0]ReadDataM,
                    input logic RegWriteW,
                    input logic [1:0]ResultSrcW,
                    output logic [31:0]PCF,
                    output logic [6:0]opD,
                    output logic [2:0]funct3D,
                    output logic funct7b5D,
                    output logic ZeroE,
                    output logic [31:0]ALUResultM,
                    output logic FlushE,
                    output logic [31:0]WriteDataM);
logic [31:0]RD1E;
logic [31:0]RD1D;
logic [31:0]RD2D;
logic [4:0]Rs1E,Rs2E,RdM,RdW,Rs1D,Rs2D,RdE;
logic [1:0] ForwardAE,ForwardBE;
logic StallF,StallD,FlushD;
logic [31:0]PCPlus4F,PCTargetE,PCRegE,PCF_p;

logic [31:0]InstrD,PCD,PCPlus4D,ResultW,ImmExtD;//RD1D,RD2D,

logic [31:0]RD2E,PCE,ImmExtE,PCPlus4E,SrcAE,SrcBE,WriteDataE;//RD1E,
logic [4:0]RdD;

logic [31:0]PCPlus4M;

logic [31:0] ALUResultW,ReadDataW,PCPlus4W;
//hazard unit
hazard_unit_pip
hazard(Rs1E,Rs2E,RdM,RegWriteM,RdW,RegWriteW,ResultSrcE0,Rs1D,Rs2D,RdE,PCSrcE,Forw
ardAE,ForwardBE,StallF,StallD,FlushE,FlushD);
//fetch stage
mux3_pip mux3(PCPlus4F,PCTargetE,PCRegE,PCSrcE,PCF_p);
flopenr_pip flopenr(clk,reset,~StallF,PCF_p,PCF);
adder4_pip adder4(PCF,PCPlus4F);
//decode stage
```

```
//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
 module datapath(input logic clk, reset,
                        input logic [1:0] ResultSrc,PCSrc,
                        input logic    ALUSrc,
                        input logic RegWrite,
                        input logic [2:0] ImmSrc,
                        input logic [2:0] ALUControl,
                        output logic Zero,
                        output logic [31:0] PC,
                        input logic [31:0] Instr,
                        output logic [31:0] ALUResult, WriteData,
                        input logic [31:0] ReadData);
            logic [31:0] PCNext, PCPlus4, PCTarget;
            logic [31:0] ImmExt;
            logic [31:0] SrcA, SrcB;
            logic [31:0] Result;
            // next PC logic
            flopr #(32) pcreg(clk, reset, PCNext,PC);
            adder4 pcadd4(PC, PCPlus4);
            adder pcaddbranch(PC, ImmExt, PCTarget);
            mux3 pcmux(PCPlus4, PCTarget, ALUResult,PCSrc, PCNext);
            // register file logic
            reg_file rf(Instr[19:15],Instr[24:20],Instr[11:7],Result,clk,RegWrite,SrcA,WriteData);
            extend_unit ext(Instr, ImmSrc, ImmExt);
            // ALU logic
            mux2 srcbmux(WriteData, ImmExt, ALUSrc, SrcB);
            alu alu(SrcA, SrcB, ALUControl, ALUResult, Zero);
            mux3 resultmux(ALUResult, ReadData, PCPlus4,ResultSrc, Result);
endmodule
```

```
//@author: Zhaoyu Wang zceezw1@ucl.ac.uk    pipeline hazard unit
module hazard_unit_pip(input logic[4:0] Rs1E,Rs2E,RdM,
                                        input logic RegWriteM,
                                        input logic[4:0]RdW,
                                        input logic RegWriteW,//forwarding
                                        input logic ResultSrcE0,
                                        input logic[4:0] Rs1D,Rs2D,
                                        input logic [4:0]RdE,
                                        input logic [1:0]PCSrcE,
                                        output logic [1:0] ForwardAE,ForwardBE,//forwarding
                                        output logic StallF,StallD,FlushE,//stall
                                        output logic FlushD);

    logic lwStall;
    //FORWARDING TO SOLVE DATA HAZARD
    always@(*)
    begin
    if((Rs1E!=0) & ((Rs1E==RdM)&RegWriteM))
            ForwardAE = 2'b10;
        else if((Rs1E!=0) & ((Rs1E==RdW)&RegWriteW))
            ForwardAE = 2'b01;
        else
            ForwardAE = 2'b00;
    if((Rs2E!=0) & ((Rs2E==RdM)&RegWriteM))
            ForwardBE = 2'b10;
    else if((Rs2E!=0) & ((Rs2E==RdW)&RegWriteW))
            ForwardBE = 2'b01;
        else
            ForwardBE = 2'b00;

    end


    //stall for lw

    assign lwStall = ResultSrcE0 & ((Rs1D == RdE)|(Rs2D == RdE));
    assign StallF = lwStall;
    assign StallD = lwStall;



    //control hazard
    always@(*)
    begin
```

```
//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
module adder(input logic [31:0] a,b,
                    output logic [31:0] y);
   assign y = a+b;
endmodule


//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
module alu(input logic[31:0] a,b,          //need modify for beq <>!=
              input logic[2:0]alu_control,
              output logic[31:0] alu_result,//modify test for zero for branch
              output logic zero);
      //assign zero = a-b;
      always_comb
      begin
            case(alu_control)
                  3'b000 : alu_result = a+b;
                  3'b001 : alu_result = a-b;
                  3'b010 : alu_result = a&b;
                  3'b011 : alu_result = a|b;
                  3'b100 : alu_result = a^b;
                  3'b101 : alu_result = a<<b;
                  3'b110 : alu_result = a>>b;
                  3'b111 : alu_result = a>>>b;
                  default: alu_result = 32'bx;
            endcase

            case(a-b)
                  32'b0 : zero = 1'b1;
                  default : zero = 1'b0;
            endcase
      end
endmodule
```

```
//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
module extend_unit(input logic [31:0] instr,
                   input logic [2:0] immsrc,
                   output logic [31:0] immext);

    always_comb
        case(immsrc)
            3'b000: immext = {{20{instr[31]}},instr[31:20]};
            3'b001: immext = {{20{instr[31]}},instr[31:25],instr[11:7]};
            3'b010: immext = {{20{instr[31]}},instr[7],instr[30:25],instr[11:8],1'b0};
            3'b011: immext = {{12{instr[31]}},instr[19:12],instr[20],instr[30:21],1'b0};
            3'b100: immext = {{20'b0},instr[31:20]};
            3'b101: immext = {{20'b0},instr[31:25],instr[11:7]};
            3'b110: immext = {{20'b0},instr[7],instr[30:25],instr[11:8],1'b0};
            3'b111: immext = {{12'b0},instr[19:12],instr[20],instr[30:21],1'b0};
            default: immext = 32'bx;
        endcase
endmodule

//@author: Zhaoyu Wang    register with reset and enable signal
module flopenr_pip #(parameter WIDTH = 32)
                   (input logic clk, reset,
                   input logic en,
                     input logic [WIDTH-1:0] d,
                   output logic [WIDTH-1:0] q);
  always @(posedge clk, posedge reset)
  if (reset) q <= 0;
  else if (en) q <= d;
endmodule
```

```
//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
module flopr#(parameter WIDTH = 32)
                (input logic clk,reset,
                input logic [WIDTH-1:0]d,
                output logic [WIDTH-1:0]q);
   always_ff@(posedge clk,posedge reset)
      if(reset) q<=0;
      else q<=d;

endmodule
//@author: Zhaoyu Wang    register with flush signal
module floprc_pip#(parameter WIDTH = 32)
                (input logic clk,reset,clear,
                input logic [WIDTH-1:0]d,
                output logic [WIDTH-1:0]q);
   always_ff@(posedge clk,posedge reset)
      if(reset) q<=0;
      else if(clear) q<=0;
      else q<= d;

endmodule
//@author: Zhaoyu Wang    register with reset, flush signal and enable signal
module flopenrc_pip #(parameter WIDTH = 32)
                        (input logic clk, reset,en,clear,
                        input logic[WIDTH-1:0] d,
                         output logic [WIDTH-1:0] q);


logic we;
logic actual_clk;
latch l(~clk,en,we);
assign actual_clk = we&clk;
always @(posedge actual_clk, posedge reset)
 if (reset) q <=    0;
 else if (clear) q <=    0;
 else if(en)q <=    d;
/*
 always @(posedge clk, posedge reset)
 if (reset) q <=    0;
 else if (clear) q <=    0;
 else if(en)q <=    d;
*/
endmodule
```

```
//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
module mux2(input logic [31:0] d0,d1,
                input logic s,
                output logic [31:0] y);
  assign y = s ? d1 : d0;
endmodule

//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
module mux2_2(input logic    d0,d1,
                input logic s,
                output logic y);
  assign y = s ? d1 : d0;
endmodule

//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
module mux3(input logic [31:0] d0,d1,d2,
                input logic[1:0] s,
                output logic [31:0] y);

  assign y = s[1]? d2:(s[0]?d1:d0);

endmodule
//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
module reg_file(input logic [4:0]RA1,RA2,WA,
                    input logic [31:0] data_in,
                    input logic clk,write_enable,
                    output logic [31:0] RD1,RD2);
    logic [31:0]rf [0:31];


    always@(posedge clk)
    if(write_enable)
        rf[WA] <= data_in;


    assign RD1 = (RA1!=0)?rf[RA1]:0;
    assign RD2 = (RA2!=0)?rf[RA2]:0;

endmodule
```

```
//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
module data_mem(input logic clk,we,
                input logic [31:0] a,wd ,
                output logic [31:0] rd);
    logic [31:0] data_MEM [63:0];//test a 64 word data_mem, edit size later

    assign rd = data_MEM[a[31:2]];//word aligneed

    always_ff@(posedge clk)
        if(we) data_MEM[a[31:2]] <= wd;

endmodule
```

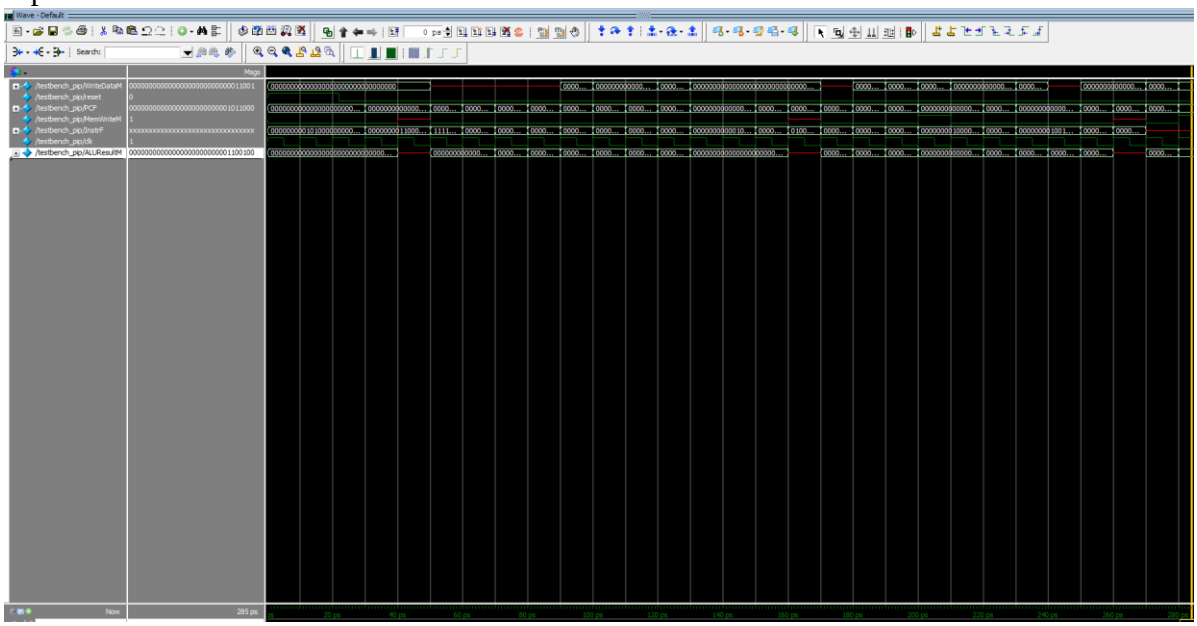
```
//@author: Zhaoyu Wang zceezw1@ucl.ac.uk//@author: Zhaoyu Wang zceezw1@ucl.ac.uk
module instru_mem(input logic [31:0] a,
                  output logic [31:0] rd);
  logic [31:0] data_MEM [63:0];
  initial
 $readmemh("riscvtest.txt",data_MEM);
 //$readmemh("fibo.txt",data_MEM);

  assign rd = data_MEM[a[31:2]];

endmodule
```

7.2 Python Script to create machine code from assembly code

```
"""
@author: Zhaoyu Wang
the riscv_assembler tool is referred from github,
an open sorce RISC-V project
the link is shown in reference 15
"""
from operator import le
from riscv_assembler.utils import *

tk = Toolkit()

instr1 = tk.I_type("addi","x0","0","x9") #addi s1,zero,0
instr2 = tk.I_type("addi","x0","1","x18")#addi s2,zero,1
instr3 = tk.I_type("addi","x0","2","x19")#addi s3,zero,2
instr4 = tk.I_type("addi","x0","7","x20")#addi s4,zero,7
instr5 = tk.SB_type("beq","x19","x20","24")#beq s3,s4,24 if branch is
taken, move to sw instruction
instr6 = tk.R_type("add","x9","x18","x5")#add t0,s1,s2
instr7 = tk.I_type("addi","x18","0","x9")#addi s1,s2,0
instr8 = tk.I_type("addi","x5","0","x18")#addi s2,t0,0
instr9 = tk.I_type("addi","x19","1","x19")#addi s3,s3,1
instr10 = tk.UJ_type("jal","-20","x0")#jal x0,20  jump back to the
branch instruction to keep looping
instr11 = tk.S_type("sw","x0","x18","4")#sw s2,(4)x0
list = []
list.append(instr1)
list.append(instr2)
list.append(instr3)
list.append(instr4)
list.append(instr5)
list.append(instr6)
list.append(instr7)
list.append(instr8)
list.append(instr9)
list.append(instr10)
list.append(instr11)

f = open("fibonacci.txt","w")
for instr in list:
    f.write(instr+'\n')
    print(instr)

f.close()
```

7.3 Timing analysis for both processors

- Pipelined Processor



- Single-cycle Processor