32-Bit MIPS Based Single-Cycle CPU Implementation of Leap Year Judgement

Jevon Wang[1]

Since the dawn of the 21st century, numerous hardware programmable languages such as Verilog and AHDL, have emerged, revolutionizing traditional digital circuit design methodologies. Accompanied by corresponding development environments and augmented with features like modularity, these languages have significantly advanced the complexity of logic circuit design. This paradigm shift mirrors the relationship between high-level programming languages and their lower-level counterparts, assembly and machine code. This paper leverages an opportunity presented by a school course to design a single-cycle CPU based on the Verilog language, featuring a simple 32-bit MIPS instruction set under a Harvard architecture.

1 Brief Introduction

The course assignment involved designing a single-cycle CPU, which was then programmed into an FPGA development board. This setup was tasked with performing functions such as data input, instruction execution, and data return. To illustrate, Let's take the leap year judgement as a case. A positive integer was inputted via a touchscreen, followed by the release of a clock signal through a button on the board, prompting the CPU to execute instructions stored in the instruction memory. The outcome, indicating whether the year was a leap year (represented by 1 for yes and 0 for no), was then displayed on the screen. The input and output processes were implemented through operations on the data memory.

Our course experiment platform utilized the LOONGSON Development Board (FPGA-A7-PRJ-UDB), equipped with features such as a touchscreen, LED lights, dip switches, and single-step debugging switches. The Vivado software was employed for programming the development board. Additionally, iverilog and GTKWave were used as cross-platform tools for development, simulation, and debugging.

1 Contact email: contact@cnily.me

2 Underlying Architecture of the CPU

The Program Counter (PC) indicates the address of the currently executing instruction in the instruction memory. The Next Program Counter (NPC) represents the output of the next instruction, acting as a pending signal. Upon receiving a clock signal, the PC module functions as a flip-flop, outputting the input NPC as the new PC value. The current value of PC retrieves the corresponding 32-bit instruction from the instruction memory, which is then fed to the instruction decoder in the CPU to generate the appropriate control signals. The simple process of an unsighed addition instruction is depicted in Figure 1. ADDU instruction is a R-type instruction, requiring 3 operands rs, rt and rd as 5-bit address of the register. The instruction decoder will output the ALU control signal as ADDU, and assign port rsc as data rs, port rtc as data rt, port rdc as rd.

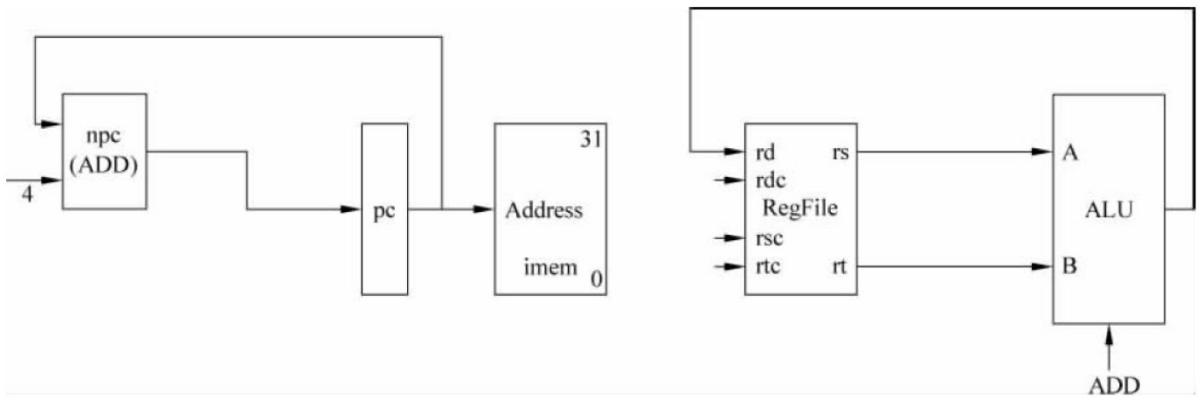


Figure 1: Process of an Instruction of ADDU[1](P250)

Module imem means instruction memory. Port rsc, rtc, rdc are three 5-bit addresses of the register, corresponding to three 32-bit data rs, rt and rd. The Arithmetic Logic Unit (ALU) receives two 32-bit operands and a control signal, outputting a result corresponding to the given control signal.

The process involving data storage and retrieval becomes more complex. Figure 2 illustrates the simplified structure of a store word (SW) I-type instruction. In the instruction memory, the three operands of the SW instruction - rs (5 bits), rt (5 bits), and immediate (16 bits) - are fed into the inputs of the register ports rsc, rtc, and the sign-extension module, respectively. The sign-extension module expands the immediate value to 32 bits, facilitating its use as a 32-bit output for the ALU module. Note that both the instruction memory and data memory are not part of the CPU structure but are crucial I/O devices in the Harvard architecture. For the SW instruction, the instruction decoder outputs enable signal and write signal to the data memory. These signals are essential as they ensure that, in the absence of data memory access operations, various input data connected to the data memory over a network do not alter the data memory.

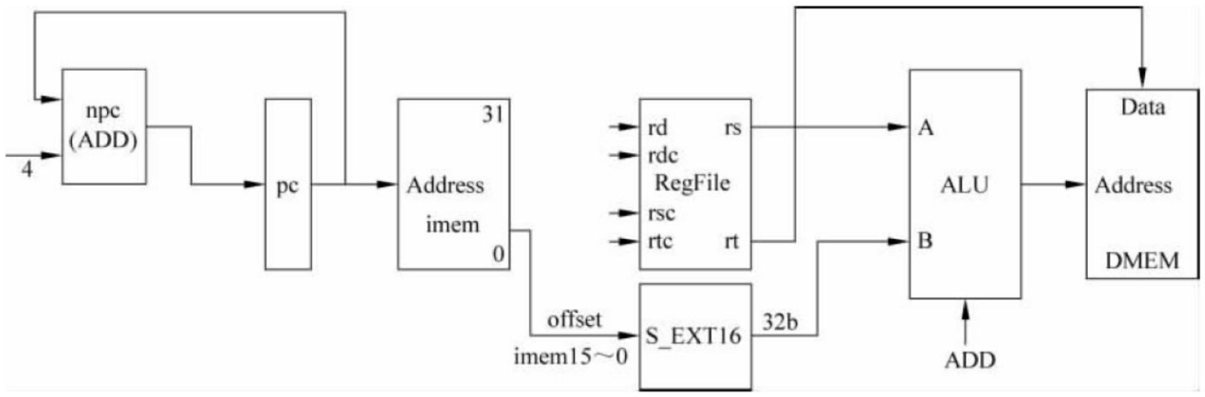


Figure 2: Process of an Instruction of SW[1](P255)

Module DMEM means data memory. The ALU adds the 32-bit register output rs with the sign-extended immediate value to generate a 32-bit data, serving as the address for the data memory. Meanwhile, the register output rt, as the actual 32-bit data to be stored, is written into the corresponding row of the instruction memory.

Another typical logic structure involves altering the input value of the PC. I-type instructions like BEQ and BNE, as well as J-type instructions like J, result in the value of next PC being different from PC + 4 bytes (32 bits, or one instruction line). Taking the J instruction as an example, as shown in Figure 3, the 26-bit immediate is left-shifted by two bits (converting bit address to byte address) and then ORed (equivalent to replacing the lower 28 bits of NPC) with the next instruction address to obtain the address of a new Next PC. A data selector MUX

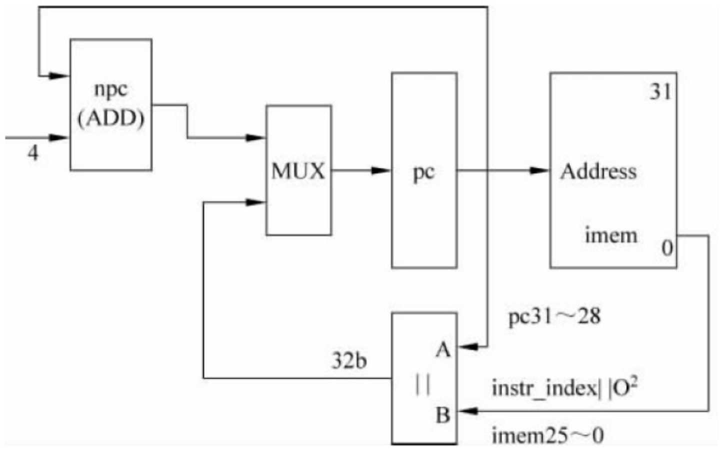


Figure 3: Process of an Instruction of J[1](P257)

The computed address along with the output from the NPC module, is selected by a data selector and then used as the input for the PC module.

chooses between two results, controlled by a signal from the instruction decoder.

Each instruction has a unique logical structure and control signals, which can be integrated to form the entirety of the CPU. By employing various data selectors and altering their select signals, as shown in Figure 4, the effective pathways can be modified to achieve the desired instruction effects. The instruction decoder, based on the 32-bit output from the instruction memory, generates appropriate module input signals and control signals through combinational logic circuits. These include ALU operation control signals, data selector control signals, data memory enable and read/write signals, etc.

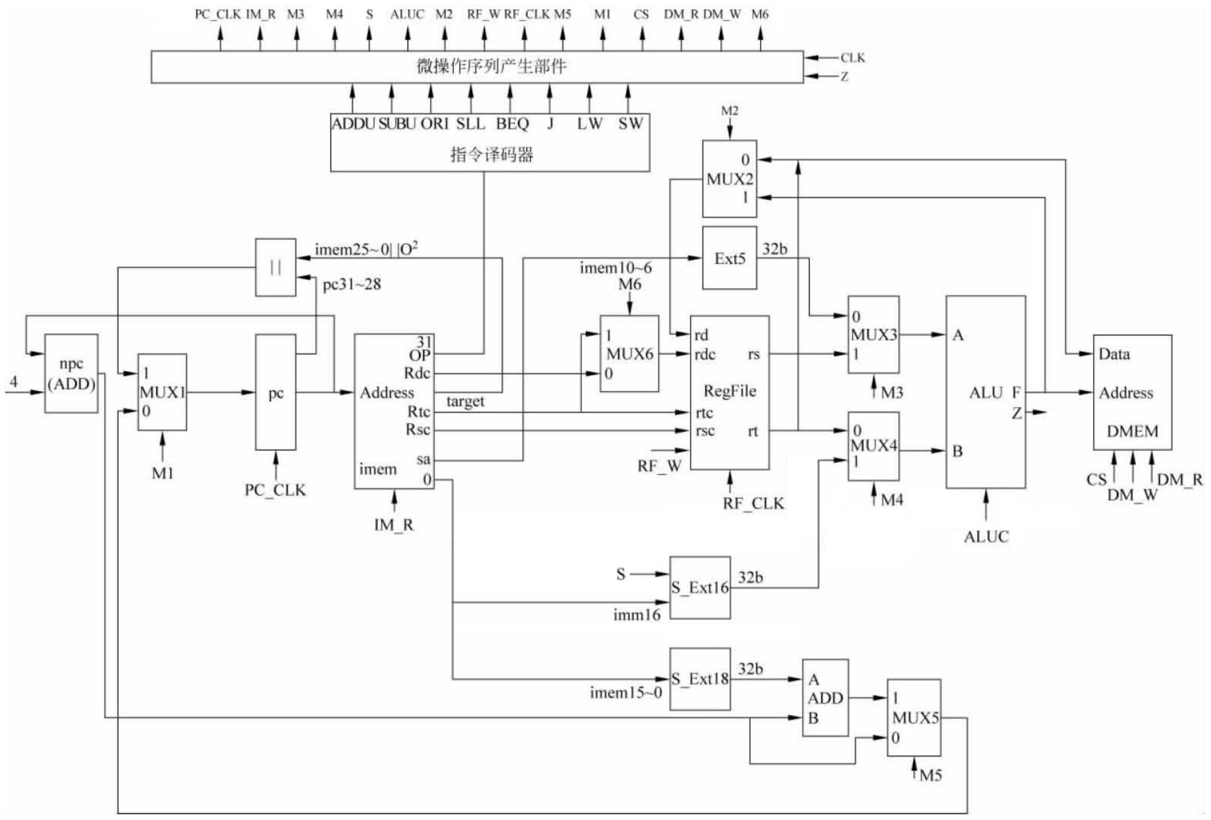


Figure 4: Full Structure of the MIPS Instruction Set, Excluding the JR, SLLV, SRLV, SRAV, LUI, SLTI, and JAL Instructions[1](P268)

3 Verilog Design

To enhance usability and extensibility, the code[1] is highly modular in its layout, with multiple parameters and macro definitions set up. For the initialization of the

1 The project is open sourced at https://github.com/cnily03-hive/single-cycle-cpu

instruction and data memories, the “readmemh” function is used to load data from files. The ALU control signals are defined separately as macros. In the top-level module, variables for the frequency division module and the debounce module are defined using parameter statements.

Regarding simulation and debugging, the simulation module is set to pause every 500 clock cycles by default. Additionally, we have developed custom scripts (named make) [1] to assist in debugging, achieving functionality similar to CMake. Furthermore, an ASM Transformer[2] was developed to facilitate the direct conversion of assembly code into machine code and corresponding hexadecimal files, with the core of this transformer integrated into the scripts named make[3].

The design of the top module fully utilizes the hardware of the FPGA development board, implementing features such as single-step debugging, continuous clock, and visualization via LED and touchscreen display. For the CPU's input clock (positive edge triggered), buttons for single-step pulse and dip switches for continuous clock (1 MHz by module FreqDiv) are set up. For the CPU's reset signal (negative edge triggered), a reset button is provided, used only to reinitialize CPU components including registers to their initial state, without resetting I/O devices like the instruction and data memories. The CPU reset and clock levels are indicated by two LEDs: a lit LED denotes a high level. In single-step debug mode, the reset level LED remains on until the reset button is pressed, while the clock level LED lights up only when the single-step debug button is pressed. Due to frequency limitations and human perceptual capabilities, when the 1 MHz continuous clock switch is on, the clock level LED appears constantly lit but dim.

The top module has macro-defined two data memory addresses whose data can be written via touchscreen, and two data memory addresses for monitoring data written by the CPU. When selecting an address and inputting data via the dip switch and touchscreen, and then clicking OK, it will generate an input pulse to the data memory and signals of enable, in which case, the data memory immediately updates its content of the corresponding address row. When the CPU attempts to write data to the data memory, the updated data at the corresponding address is also displayed

1 See make (shell script) at the root directory of the project source code.
2 See at https://asm.tampoo.io
3 See transformer.js at the root directory of the project source code.

on the touchscreen, which is functionality achievable through counters and logical OR operations. These functions greatly facilitate instruction debugging.

The top module enables the touchscreen to display the currently executing instruction and its address (PC), as well as the values of the 32 registers. During testing, we observed jitter presenting by once pressing single-step debug button, causing multiple level changes in a short time and resulting in jump changes in the PC and instruction values on touchscreen, which is disadvantageous for debugging. Therefore, a submodule was added to provide a debounce of default 10 milliseconds.

The pins corresponding to the buttons, dip switches, and LEDs utilized for these functionalities are defined in the constraints.xdc file. Their actual locations on the development board are depicted in Figure 5, and their corresponding functions are detailed in Table 1.

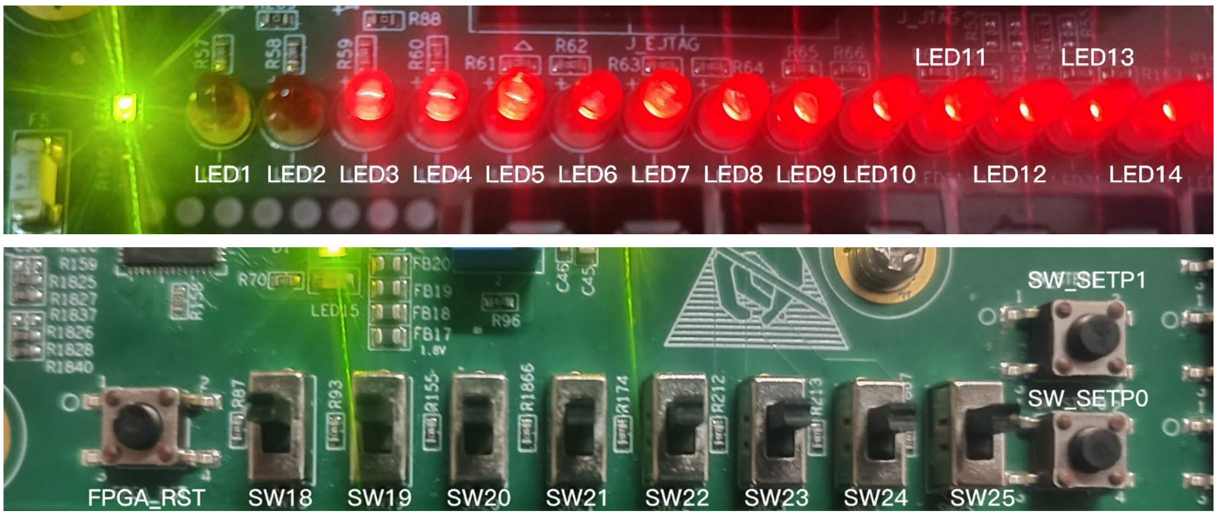


Figure 5: Pin-Identifier Correspondence Diagram on LOONGSON FPGA Board

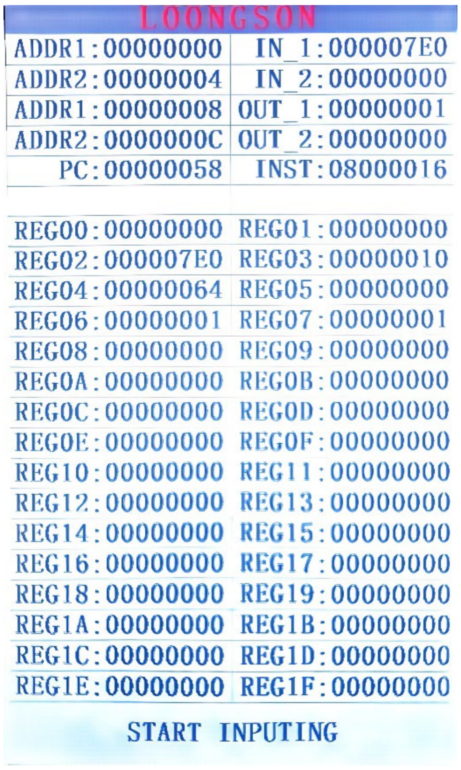
Table 1: Pin-Identifier Correspondence Function Table

Identifier	Pin	Description
SW18	AC21	Select input address (up for address 1, down for address 2)
SW20	AC22	Slide down to give continuous clock
SW_STEP0	Y5	Press to give one CPU clock (button debounced)
SW_STEP1	V6	Press to reset the CPU (button debounced)
FPGA_RST	Y3	Reset the touch screen (only clear input but not memory)
LED1	H7	The reset signal (lit when high level)
LED2	D5	The clock signal of CPU (lit when high level)

4 Implementation of Assembly Execution

Take the judgement of whether a given year is a leap year as an example. we designate the first row of the data memory (address 0x00000000) for input data. After executing the instructions, the CPU stores the result in the third row of the data memory (address 0x00000008).

In writing the assembly code, it's important to note that since the CPU's PC defaults to 0x00000000 upon initialization, it will load the next PC value upon receiving a clock pulse and proceed to execute the next instruction. Therefore, the instruction in the first row will not be executed. We need to occupy it with a NOP (No Operation) instruction. As the simplicity of this CPU, we reserve register $0 as the $zero register. Additionally, at the end of the assembly program, we set a J instruction to load the Next PC value as the current PC value, in order to create an infinite loop, which helps debug, and we can easily observe when the program execution is complete. The relevant assembly code is provided in Appendix 1.



```
LOONGSON
ADDR1:00000000   IN_1:000007E0
ADDR2:00000004   IN_2:00000000
ADDR1:00000008  OUT_1:00000001
ADDR2:0000000C  OUT_2:00000000
   PC:00000058   INST:08000016

REG00:00000000  REG01:00000000
REG02:000007E0  REG03:00000010
REG04:00000064  REG05:00000000
REG06:00000001  REG07:00000001
REG08:00000000  REG09:00000000
REG0A:00000000  REG0B:00000000
REG0C:00000000  REG0D:00000000
REG0E:00000000  REG0F:00000000
REG10:00000000  REG11:00000000
REG12:00000000  REG13:00000000
REG14:00000000  REG15:00000000
REG16:00000000  REG17:00000000
REG18:00000000  REG19:00000000
REG1A:00000000  REG1B:00000000
REG1C:00000000  REG1D:00000000
REG1E:00000000  REG1F:00000000

        START INPUTING
```

Figure 6: Executing Result Inputting 0x000007E0 as a year to judge, and the CPU returns 0x00000001 (true) as the result.

Taking the input of year 2016 (hexadecimal 0x7E0) as an example, after resetting the CPU and turning on the continuous clock dip switch, the CPU quickly completes its execution. The results displayed on FPGA board are shown in Figure 6.

5 Summary and Outlook

Through the example above, we have essentially completed the design of a single-cycle CPU, and consequently developed additional projects such as the ASM Transformer. These development tools will better assist us in refining instructions and designing multi-cycle CPUs. Building on this project, we will further conduct research and design efforts to ultimately realize a multi-cycle 64-bit CPU under von Neumann architecture, and explore various classic even novel optimization methods.

References

[1] 张冬冬, 王力生, 郭玉臣. 数字逻辑与组成原理实践教程[M]. 清华大学出版社, 2018.

Appendix

1 ASM Code of Leap Year Judgement

```
    sll   $zero, $zero, 0         ; nop

isLeap(int):
    addi  $s0, $zero, 0           ; start DM address = 0, store to $s0
    lw    $s1, 0($s0)             ; year = DM+0
    add   $s2, $zero, $s1         ; n = year
    addi  $s3, $zero, 100         ; m = 100
    addi  $s4, $zero, 0           ; cnt = 0
    addi  $s5, $zero, 0           ; res = 0
    andi  $t0, $s1, 3             ; tt = year & 3
    bne   $t0, $zero, SAVE        ; if tt != 0, goto SAVE
    j     LEAP_1                  ; goto LEAP_1

LEAP_0:
    addi  $s4, $s4, 1             ; cnt = cnt + 1

LEAP_1:
    sub   $s2, $s2, $s3           ; n = n - m
    sra   $t0, $s2, 31            ; tt = n >> 31
    beq   $t0, $zero, LEAP_0      ; if tt == 0 (n >= 0), goto LEAP_0
    andi  $s4, $s4, 3             ; cnt = cnt & 3
    add   $s2, $s2, $s3           ; n = n + m
    beq   $s2, $zero, N_IS_0      ; if n == 0 (can mod 100), goto N_IS_0
    addi  $s5, $zero, 1           ; res = 1
    j     SAVE_0                  ; goto SAVE

N_IS_0:
    bne   $s4, $zero, SAVE        ; if cnt != 0 (cannot mod 400), goto SAVE
    addi  $s5, $zero, 1           ; res = 1

SAVE:
    sw    $s5, 8($s0)             ; DM+8 = res

END:
    j     END                     ; goto END
    sll   $zero, $zero, 0         ; nop
```

2 Hexadecimal Instruction File

```
00000000
20010000
8C220000
00021820
20040064
20050000
20060000
30470003
14E0000C
0800000B
20A50001
00641822
00033FC3
10E0FFFC
30A50003
00641820
10600002
20060001
08000015
14A00001
20060001
AC260008
08000016
00000000
```