Comp Org Project

Matthew Bui, Justin Kim, Aaron Verkleeren , Danny Zou

December 9, 2023

1 Introduction/Overview

Our group project delves into the design of a basic accumulator-based computer system, designated as Design 1 as per instructions. Our objective is to address the core question of how computers compute, using a minimalistic design approach that maintains performance efficiency. In accordance with the Seven Great Ideas in Computer Architecture, we have ensured our design encapsulates essential features.

The subsequent sections of this report will discuss Design 1's enhancements, introducing data caching and optional advanced features such as separate memory for instructions and data, as well as a pipelined datapath. The report also contrasts Design 1 with Design 2, a distinct architecture with its unique specifications. Our methodology is grounded in Verilog implementations and benchmark testing through assembly language programs, specifically crafted to assess and compare the performance of the designed architectures.

2 Design 1

Team ID: 00130

130 % 32 = 2

Team ID / Word size(bits) / Main memory size(bytes) / Main memory organization / Max num of bits used by L1 cache / Additional addressing mode

2 || 16 || $64Ki$ || $8Ki$ x 8 | $40,000$ || $Indirect$

General policies:

For our Design 1, we decided to have some general policies that would apply to all parts of our code just based on our assigned requirements.

Since our word size was 16 bits, we decided to have the first 4 bits denote the operation, and the last 12 to be used for direct addressing.

ALU:

After defining our op codes we decide to go for some of the extra credit functions as well.

Ram:

Here we followed the general coding procedure taught to us and implemented a singular ram module with line size 12 (to fit our configuration) and a then built a large module to fit out memory requirement of 64KB max memory with a structure of 8KB x 8.

CPU:

Here we implemented our cpu just as a test module, which builds all the lines into memory and then goes through them and executes the procedures by each line.

Our clock rate was also determined to be 21ns as that was the time of our slowest procedure (ie DRAM and 1 register).

Here we can see that the code correctly finds 59 as the F11 number (59 is in hexa, equals 89 in decimal), (90 is the index of the next item 59 is the item at the end).

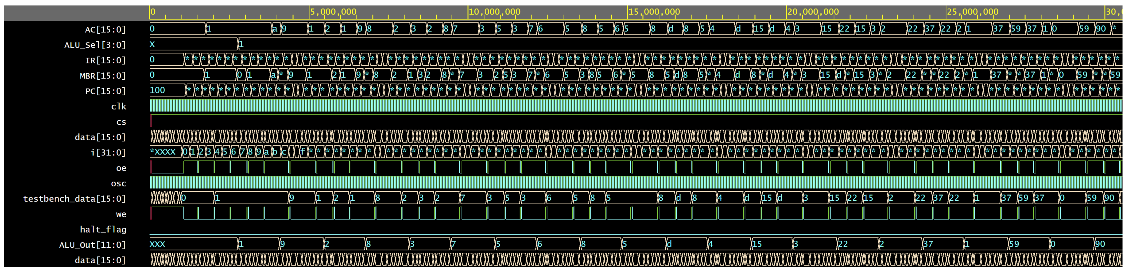


Figure 1: Design 1 tested on fib sequence

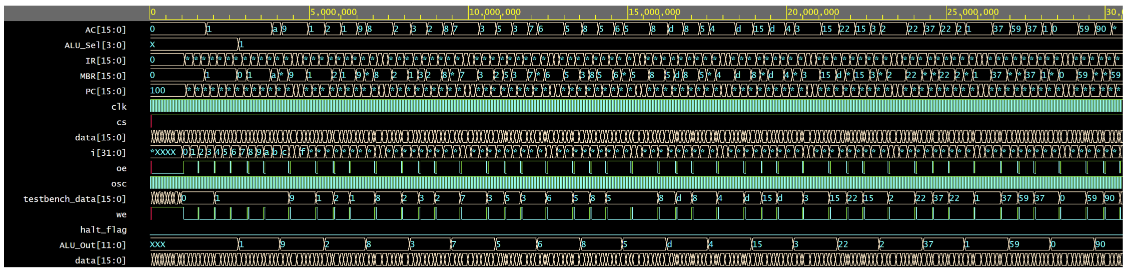
3 Improvements to Design 1

3.1 Design 1a

Assume same conditions as Design 1 with improvements

Included new bit after the four bits of op code to decide if the address is direct (0) or indirect (1).

If address is indirect, just call the memory addressing again to get the actual value, then preform the operation as necessary.



Here we can see that design 1a builds upon design 1 and correctly finds 59 to be the value of the F11 number.

3.2 Design 1b

New Clock Rate = 2ns (because of the switch from DRAM to SRAM for the cache) Team ID: 00130

130 % 4 = 2

Architecture = General purpose register
Addressing modes = Register addressing, based addressing, register indirect addressing

Per the Appendix A requirements for 2, I implemented it as a direct mapped cache because it was the simplest and efficient enough for our needs. Each cache line stores data from the main memory, along with metadata such as a tag and a valid bit. The tag is a portion of the main memory address used to identify which block of memory is stored in the cache line. The memory addresses are divided into three parts: tag, index, and block offset. The index determines which cache line an address maps to, the tag is used to check if the correct data is in the cache line, and the block offset locates the specific data within the cache line.

4 Implementation of our designs

5 Simulation and testing

Computing Fibonacci number F_{11} by using a loop.

Extra Credt - Matrix multiplication of two 8 by 8 matrices.

```
.data
    matrixA: .space 256  # 8x8 matrix, 4 bytes per element
    matrixB: .space 256  # 8x8 matrix, 4 bytes per element
    matrixResult: .space 256  # 8x8 matrix, 4 bytes per element

.text
.globl main
main:
    # Load base addresses of matrices into registers
    la $t0, matrixA        # $t0 - base address of matrixA
    la $t1, matrixB        # $t1 - base address of matrixB
    la $t2, matrixResult   # $t2 - base address of matrixResult

    # Initialize loop counters (i, j, k, respectively)
    li $s0, 0
    li $s1, 0
    li $s2, 0

    # i loop
i_loop:
    bge $s0, 8, end_i_loop     # if i >= 8, exit loop

    # j loop
    li $s1, 0              # reset j to 0
j_loop:
    bge $s1, 8, end_j_loop     # if j >= 8, exit loop

    # Reset sum to 0
    li $t3, 0              # $t3 - sum for the dot product

    # k loop
    li $s2, 0              # reset k to 0
k_loop:
    bge $s2, 8, end_k_loop     # if k >= 8, exit loop

    # Calculate addresses for the current elements
    mul $t4, $s0, 32       # row offset for matrix A
    add $t4, $t4, $s2
    sll $t4, $t4, 2        # byte offset
    add $t4, $t4, $t0      # address of A[i][k]

    mul $t5, $s2, 32       # row offset for matrix B
    add $t5, $t5, $s1
    sll $t5, $t5, 2        # byte offset
    add $t5, $t5, $t1      # address of B[k][j]
    # Load current elements and multiply
    lw $t6, 0($t4)         # load A[i][k]
    lw $t7, 0($t5)         # load B[k][j]
    mul $t6, $t6, $t7      # multiply
    add $t3, $t3, $t6      # add to sum

    # Increment k
    addi $s2, $s2, 1
    j k_loop

end_k_loop:
    # Calculate address for the result element
    mul $t4, $s0, 32       # row offset for matrixResult
    add $t4, $t4, $s1
    sll $t4, $t4, 2        # byte offset
    add $t4, $t4, $t2      # address of matrixResult[i][j]

    # Store the sum into the result matrix
    sw $t3, 0($t4)         # store sum at matrixResult[i][j]

    # Increment j
    addi $s1, $s1, 1
    j j_loop

end_j_loop:
    # Increment i
    addi $s0, $s0, 1
    j i_loop

end_i_loop:

    # Exit program
    li $v0, 10             # Exit system call
    syscall
```

Extra Credit - Recursive Implementation of factorial function
I believe implementing a recursive factorial function will be impossible with our ISA because of several things. First of all, we are not implementing a hardware-supported stack, which is crucial for managing the return addresses and local variables of recursive function calls. Without a stack, it becomes extremely different to keep track of each recursive call's state. Second, because of our very limited number of instructions, crucial operations like "call" or "return" which are used to handle recursion are not available, causing a lack of control structures needed for recursion. We also have to be wary of memory constraints. So that's why I implemented it iteratively.

```
    Address   Value     Code
100       1126         Load        N
102       9400         Skipcond    400
104       A10C         Jump        Factorial
106       112C         Load        Neg1
108       2128         Store       Result
10A       A124         Jump        End
10C       1126       Factorial, Load   N
```

```
10E       212A          Store      Ctr
110       112C          Load       Neg1
112       2128          Store      Result
114       1128       Loop,      Load   Result
116       712A          Mult       Ctr
118       2128          Store      Result
11A       112A          Load       Ctr
11C       312C          Add        Neg1
11E       212A          Store      Ctr
120       9400          Skipcond   400
122       A114          Jump       Loop
124       7000       End,       Halt
126       0003      N,       Dec 3
128       0000      Result,  Hex 0
12A       0000      Ctr,     Hex 0
12C       FFFF      Neg1,    Dec -1
```

5.1 Benchmarking

Computing Fibonacci number with loop machine code translation

```
Address  Value  Code
        ORG 100                / Fibonacci sequence calculation
100     1120   Load    First   / Load the first number (0) into AC
102     212A   Store   Fibo    / Store it in the Fibonacci result
104     1122   Load    Second  / Load the second number (1) into AC
106     2124   Store   Next    / Store it as the next number to add

loop
108     1124 load next : Load bigger number
10a     2122 store second : temp holder
10c     312a add Fibo : AC = next number
10e     2124 store next : store in next
110     1122 load second : load temp
112     212a store fibo : store in fibo
114     1126   Load    Counter / Load the loop counter
116     3128   Add     DecOne  / Decrement the counter
118     2126   Store   Counter / Store the new counter value
11A     9200   Skipcond 400    / If counter is 0, end the loop
11C     A108  Jump    @LoopAddr    / Jump back to the start of the loop
11E     FFFF   Halt            / End the program

120     0000   First,  Dec 0   / The first Fibonacci number
122     0001   Second, Dec 1   / The second Fibonacci number
124     0001   Next,   Dec 1   / The next Fibonacci number initially set to the same as Second
126     000A   Counter,Dec 10  / Loop counter, for 10 iterations (since we already have the first two nu
128     0FFF   DecOne, Dec -1  / Used for decrementing the counter
12A     0000   Fibo,   Hex 0   / Fibonacci number storage
12C     0103   LoopAddr, Hex 103 / Address of the start of the loop
```

6 References

Konstantin, K. (2023, December 4). Meet virtually with Cisco Webex. anytime, anywhere, on any device. Cisco Webex Site. https://rensselaer.webex.com/recordingservice/sites/rensselaer/recording/b26f496f751d103cb7fe22c98a2a5

Konstantin, K. (n.d.). CSCI 2500 — Computer Organization: How do computers actually compute?. Submitty CSCI 2500. https://submitty.cs.rpi.edu/courses/f23/csci1100/course_materials

Admin. (n.d.). Verilog Tutorial. ChipVerify. https://www.chipverify.com/tutorials/verilog

Tala, D. K. (1970, February 1). Verilog Tutorial. ASIC World. https://www.asic-world.com/verilog/veritut.html

GeeksforGeeks. (2023, July 14). Fibonacci series program in C. GeeksforGeeks. https://www.geeksforgeeks.org/c-fibonacci-series/