Replicating MIPS Principles: Custom CPU and Assembler

Harrington, Collin

CSC 270

Union College

Professor Chris Fernandes

I. Introduction

MIPs architecture is a common standard in the study of computer architecture. For this reason, we investigate features of MIPS CPU design and implement them into a custom CPU in Logisim as well as writing an assembler for this CPU in Python. The CPU implements R-type instructions, I-type instructions, branching, jumping and the assembler supports labels and multi file linking. In section II we discuss these design decisions in more detail. In section III we discuss our instruction word format. In section IV we introduce our verification process. In section V we include the results of verification. In section VI we discuss the results. In the final section, section VII, we include directions and comments about downloading and running our design.

II. Design Decisions

In our design we selected a 16-bit CPU, with 16 registers each able to hold words that are two bytes in size. This necessitated that our RAM addresses are at maximum 16 bits and that each word of ram is also two-bytes. The 16-bit ALU included uses a standard ripple carry adder. In order to represent three registers and all of our included operations, a 21 bit instruction word was chosen, and will be discussed in more detail in section III. We implemented branching and jumping by labels along with multi-file linking in order to assist in the construction of more complicated program structures. Labels can be local or global, assumed by whether or not they are referenced within a file. Local labels are handled by appending the containing file's name at assemble time. This means that naming a label in "main.asm" "other_file_loop.asm" in a linkage with the file "other_file" containing the local label "loop" can cause errors in program execution. There are no pseudonyms for branching instructions such as "blt" or ability to parse .data headers in assembly files. Furthermore there is no support for a stack pointer and function based jump calls. It is, however, still possible to implement these behaviors within a program with a lot of effort from the programmer. The register $0 is often assumed to have the value 0 within, however, nothing guarantees this behavior in this CPU.

III. Instruction Word

Our CPU design uses a 21 bit instruction word. A breakdown of the different instruction word formats is displayed in Table I. Foremost in our instruction word is the mode and operation segments. The mode segment is three bits that designate whether or not the instruction is an I-type instruction or a R-type instruction. When the mode bits are all zeroes, the instruction is R-type, otherwise, it is I-type, and the mode bits determine the behavior of our datapath. Both the R-type and I-type instructions use the operation bits to determine the function of the ALU. When a jump instruction is presented (mode = 0b100) then the rest of the instruction word following is processed as an immediate specifying the jump target. Consult Appendix A for a full breakdown of the I-type modes and operation codes.

TABLE I
Instruction Word Format

	3b	3b	4b	4b	4b	3b
R-Type Instructions	Mode: 0b000	Op Code:	$dest	$src1	$src2	PADDING 0b000
I-Type Instructions	Mode:	Op Code:	$dest (or $src2)	$src1 (or $base)	Imm	
J Instruction	Mode: 0b100	Target program memory address				

For instance consider the assembly instruction “beq $3 $5 8”. To translate this instruction we would find the “beq” entry in Appendix A and see that the mode and op code are <101> and <011> respectively. Then the literal access registers $3, and $5 are resolved to 4b numbers

<0011> and <0101> respectively. Finally, the offset “8” is resolved to a 7b number <0001000>. Each of these are appended in order to form the instruction word 0b101011001101010001000, in hex 0x159A88.

IV. Verification Process

Our verification proceeds via the tests listed in Table II. All of the files listed can be found at [1] in the “asm/test” directory.

TABLE II
Verification and Test Files

Test(s)	Expected Result
Basic-R-Type	Registers [1..7] contains [1..7]
Basic-LW-SW	Registers [1..7] contains [7..1] RAM [1..7] contains [1..7]
test_BEQ test_BEQ_labels test_BNE test_BNE_labels	Registers 4 and 5 contain 4 and 5 respectively
test_BEQ_negativeOffset test_BEQ_negativeOffset_labels	Registers [1..5] all contain 5
test_jump_absolute test_jump_labels	Register 1 contains 3
test_SLT	Registers 3 and 4 contain 1 Registers 5 and 6 contain 0
linking_test_a + linking_test_b	Registers 1..3 all have 4
linking_test_b + linking_test_a	Registers 1..2 have 4, register 3 has 2

To verify each of these tests, an “empty.hex” file, which can be found at [1] in the “hex” directory, is uploaded to the CPUs instruction memory in Logisim. The clock is then turned high and low until the program counter points to a 0d0 line in the instruction memory. The relevant .asm files are passed to the assembler to create an “output.hex” file which is run using the same process as in “empty.hex”. At this point, screenshots are collected of the relevant registers and RAM addresses. A “+” between two file names indicates that the two files are linked using the assembler to create a single “output.hex” file. When this is the case, linking order matters. Optionally, each of the test files listed in Table II have already been converted to .hex files in “hex/test/” at [1]. For further information on how our assembler is used to create “output.hex” files, see the README.md at [1].

V. Results

For the Basic-R-type tests the results in figure 1 match the expected results.

For Basic-LW-SW tests the results in figure 2 match the expected results.

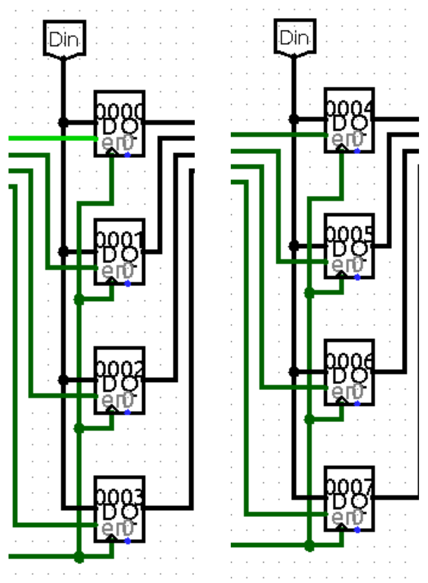

Fig. 1 Registers 0..3 (left) and 4..7 (right) after running Basic-R-Type test.

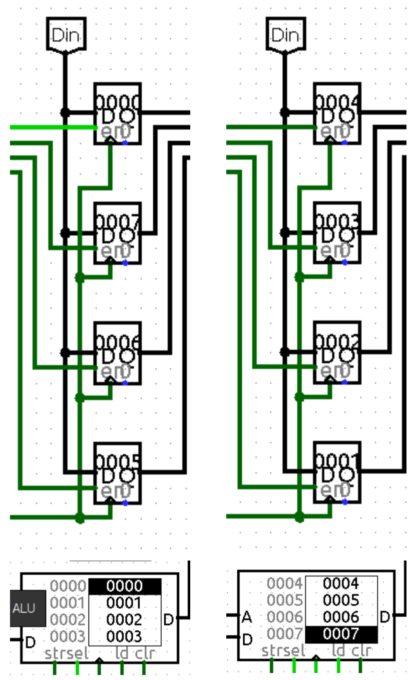

Fig. 2 Registers 0..3 and RAM 0..3 (left) as well as registers 4..7 and RAM 4..7 (right) after running Basic-LW-SW test.

For test_BEQ and test_BEQ_labels the results match the expected results.

For test_BNE and test_BNE_labels the results match the expected results.

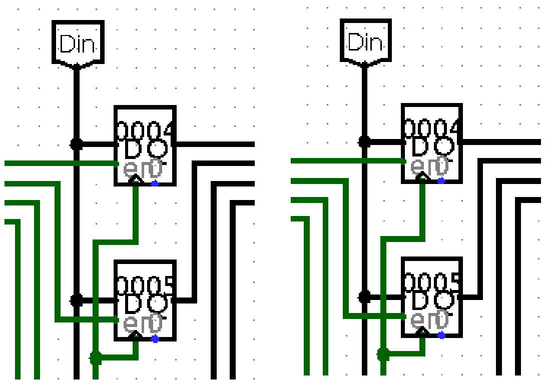

Fig. 3 Registers 4 and 5 after test_BEQ (left) and test_BEQ_labels (right).

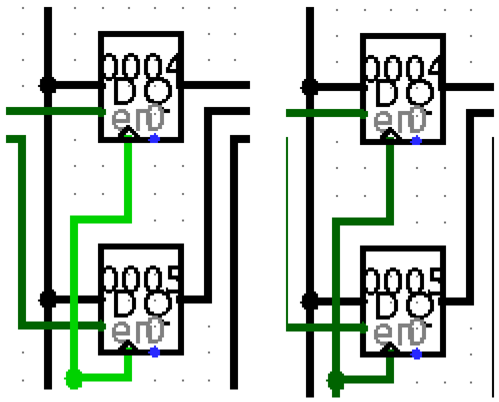

Fig. 4 Registers 4 and 5 after test_BNE (left) and test_BNE_labels (right)

For test_beq_negativeOffset and test_beq_negativeOffset_labels the results in fig 5 and fig 6 match the expected results.

For test_j_absolute and test_j_labels the results in fig 7 match the expected results.

For test_SLT the results in fig 8 match the expected results.

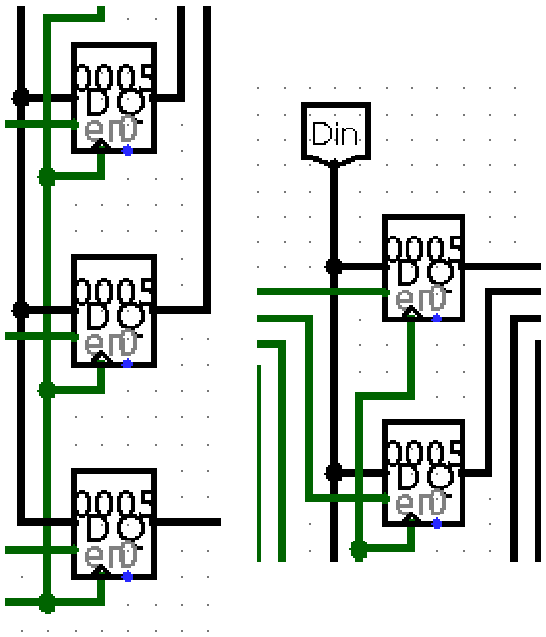

Fig. 5 Registers 1..3 (left) and 4..5 (right)

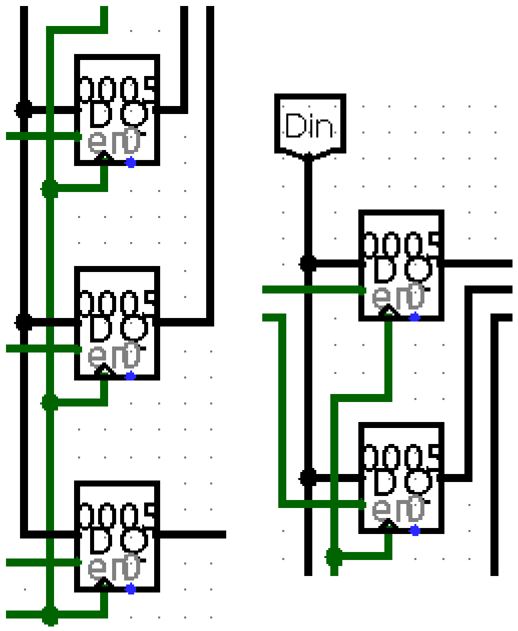

Fig. 6 Registers 1..3 (left) and 4..5 (right)

after running test_beq_negativeOffset. after running test_beq_negativeOffset_labels.

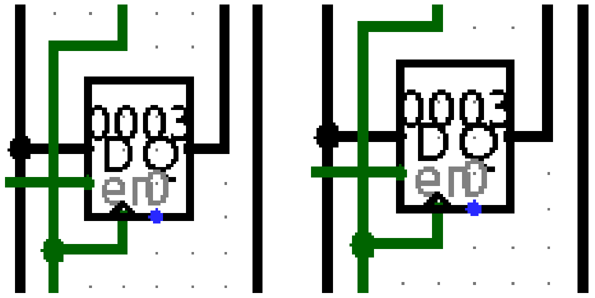

Fig. 7 Registers 1 after test_jump_absolute (left) and test_jump_labels (right)

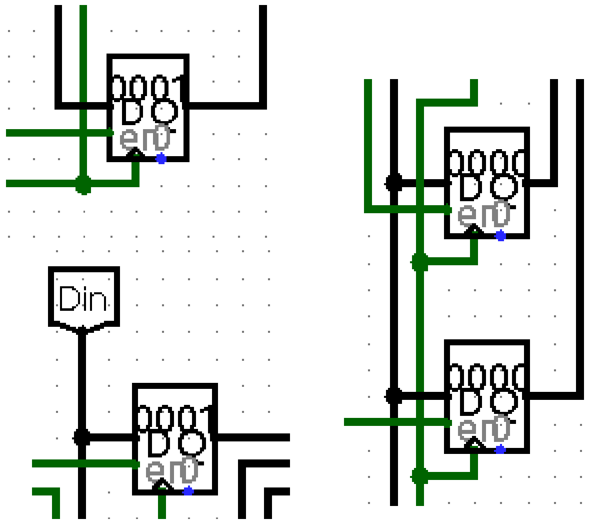

Fig. 8 Registers 3..4 (left) and 5..6 (right) after running test_SLT.

For linking_test_a + linking_test_b and linking_test_b + linking_test_the results in fig 9 match the expected results.

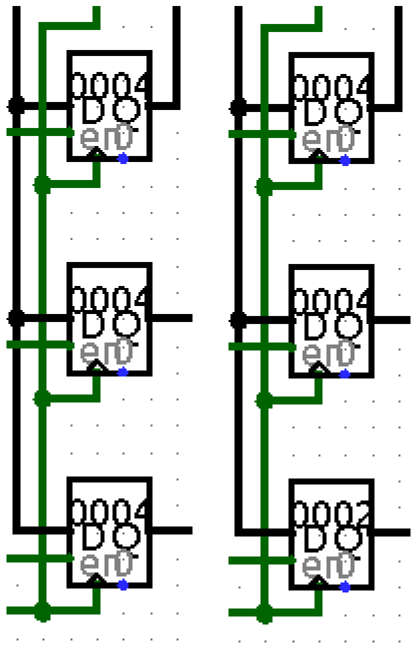

Fig. 9 linking_test results with a + b order (left) and b + a order (right)

VI. Discussion and Grade Sheet

In our results, all of our implemented features pass their respective tests. For this reason we are requesting near maximum points for each feature. A breakdown of the points requested by feature is pictured in Table III. The total requested points is 114 out of 115 points for the assignment.

Feature Implemented	Requested Points	Reasoning
16-bit CPU	(700) + 5 / 5	There are two .circ files at [X], both are 16-bit CPUs and all the “Basic” tests were validated.
BEQ	5 / 5	All of the BEQ tests have been validated, including negative offsets.
BNE	4 / 5	All of the BNE tests have been validated, but not negative offsets were not tested individually (-1). The offset is expected to work the same as for BEQ however.
Jump (j) instructions	10/10	All of the jump tests have been validated.
SLT	5/5	All of the SLT tests have been validated.
Assembler w/ labels	5/5	All of the “labels” tests have been validated.
Linking of multiple asm files	10/10	The “linking” tests have all been validated.

VII. Using this CPU and Assembler

All the material for this CPU is uploaded at [1]. The README.md there includes further directions for using the assembler as well.

References

[1] C. Harrington, Custom Assembler, (2025), GitHub repository,

Accessedhttps://github.com/charrington1521/CustomAssembler

Appendix A

Assembly Code to Machine Code: R-Type

Assembly Instruction	Corresponding Machine Code
add [$dest] [$src1] [$src2]	<000><000><$dest (4b)><$src1 (4b)><$src2 (4b)><000>
and [$dest] [$src1] [$src2]	<000><001><$dest (4b)><$src1 (4b)><$src2 (4b)><000>
or [$dest] [$src1] [$src2]	<000><010><$dest (4b)><$src1 (4b)><$src2 (4b)><000>
sub [$dest] [$src1] [$src2]	<000><011><$dest (4b)><$src1 (4b)><$src2 (4b)><000>
slt [$dest] [$src1] [$src2]	<000><100><$dest (4b)><$src1 (4b)><$src2 (4b)><000>

Assembly Code to Machine Code: I-Type

addi [$dest] [$src1] [imm]	<111><000><$dest (4b)><$src1 (4b)><imm (7b)>
andi [$dest] [$src1] [imm]	<111><001><$dest (4b)><$src1 (4b)><imm (7b)>
ori [$dest] [$src1] [imm]	<111><010><$dest (4b)><$src1 (4b)><imm (7b)>
subi [$dest] [$src1] [imm]	<111><011><$dest (4b)><$src1 (4b)><imm (7b)>
lw [$dest] [offset]([$base])	<001><000><$dest (4b)><$base (4b)><imm (7b)>
sw [$src] [offset]([$base])	<010><000><$src2 (4b)><$base (4b)><imm (7b)>
beq [$src1] [$src2] [offset]	<101><011><$src2 (4b)><$src1 (4b)><imm (7b)>
bne [$src1] [$src2] [offset]	<110><011><$src2 (4b)><$src1 (4b)><imm (7b)>
j [pc]	<100> <pc (18b)>