Computer Organization

Project 1

Jenna Uba

March 23, 2018

**Results**

The final dlx assembler program was successful. It correctly takes in an input file containing assembly language. It also correctly reads and parses through the file and distinguishes the tokens and labels. The program is able to determine which type of instruction (load, store, math, etc.) is being referenced and then it correctly executes the proper encoding sequence using bitwise operations. Finally, the program successfully writes the final hexadecimal machine code to the output file. Issues that may stand with the program are concerning efficiency. The implementation of partitioning the different instructions into separate arrays is a source for the decreased efficiency. All of these arrays are all of a size of twenty-five, however most of them do not need this much space, meaning the excess space are null values. This is not just a waste of memory space, but it now causes a longer run time because the program is looping through excess indexes. The reason each array is initialized to twenty-five is to prevent out-of-bound errors when looping through the arrays. This issue could be resolved by placing all the instructions in one array, rather than separate one. Although efficiency is a negative factor to the final program, it does operate as specified and correctly computes the hexadecimal machine code values of the inputted assembly file, making the project a success.

**Functions**

int main()

The Main Function is called at the program startup. This function initializes the output file, a vector which holds each parsed token, a vector which is used to temporarily hold each single line of instructions, a vector that holds the labels, and a vector that holds the labels locations. The main function then calls the function readParse() to read and parse the input file, as well as the encode function to begin the instruction encoding. At the end the main function returns 0 to indicate the program run has been completed.

void readParse(vector<string>\* res, vector<string>\* labels, vector<int>\* location)

The readParse function opens, reads, and parses the input file. It takes in three parameters, a pointer to the vector that holds each token in the file, a pointer to the vector that hold the label names, and a pointer to the vector that holds the location of the labels. The function loops through the file and utilizes the strtok() which is in the c standard library. This tokenizes the file by breaking into a new string whenever a comma, space, new line, tab, pound sign, or parenthesis occurs. The lower function is then called to change any uppercase characters in the token to lowercase. The readParse function then checks if the token has a colon in the last index. If it does that means the token is a label, so it is added onto the label vector and the counter keeping track of the file line is added to the location vector. If there is no colon at the end of the token, it is then added onto the token vector.

string lower(string s)

The lower function converts any characters in a string that are uppercase to lowercase. It takes in one string parameter. The function loops through each character of the string and if the isupper(char c) function returns true, the character is converted to lower case. When the function is complete it returns the new lowercase string.

void encode(vector<string>\* res, vector<string>\* type, vector<string>\* labels, vector<int>\* location, ofstream\* outFile)

The encode function determines the instruction names for and then calls the correct function to encode that instruction. It takes in five parameters: a reference to the vector holding the tokens, a reference to the vector to temporarily hold each piece of an instruction line, a reference to the vector holding the label names, a reference to the vector holding the label locations, and a reference to the output file. Inside the encode function there is a separate array holding the instruction names for load instructions, the store instructions, the math immediate instructions, the branch instructions, the ALU instructions, the jump instructions, and the jump immediate instruction. For each of these there is a parallel array holding the opcodes for the instruction names. There is also an int holding the opcode for an LHI instruction, an int to temporarily hold an opcode value, and a counter to keep track of the location of the tokens. There encode function then loops through each element of the vector holding the tokens, if the token is equal to LHI it then places each element of that instruction line into the temporary vector. Then a function is called to perform the encoding for LHI type instructions. After this function is called the temporary vector is cleared so that is can be used for the next instruction, and the counter keeping track of the token location is increased. Then the encode function nest a for loop into the current one, to loop trough all of the arrays holding the instruction names. A series of if statements are used to perform similar operations as the function for the LHI type did, however it the int that was created to hold the instruction opcode is set to the matching opcode in the opcode arrays. This instruction int is passed to the function that is called to perform the actual encoding.

void load(vector<string> type, int instr, ofstream\* outFile)

The load function is called in the encode function whenever the token is a load type. This takes in three parameters, the vector that is temporarily holding each piece of the instruction line, an int representing the instruction opcode, and a reference to the output file. Inside the load function there is a string to represent rs1. This string is initialized to the last element of the token in the third index of the temporary vector. There is also a string to represent rs2. This string is initialized to the last element of the token in the first index of the temporary vector. Then there is a string to represent the immediate value. This is initialized to the token in the second index of the temporary vector. Then there are ints that use the atoi() function, from the standard c library, to convert the rs1 string, the rs2 string, and the immediate string to an int. The int storing the immediate value is anded with the hex value 0x0000FFFF, to account for sign extension. Then the int representing rs1 is shifted to the left 21 bits and the int representing rs2 is shifted to the left 16 bits. Then the opcode, which was passed in through the parameters, gets shifted to the left 26 bits and ored with the shifted rs1 int, the shifted rs2 int, and the sign extended immediate value. This is all stored into the code int, which is then converted to hex and written to the outFile using ‘0’ as a padding.

void store(vector<string> type, int instr, ofstream\* outFile)

The store function is called in the encode function whenever the token is a store type. This takes in three parameters, the vector that is temporarily holding each piece of the instruction line, an int representing the instruction opcode, and a reference to the output file. Inside the store function there is a string to represent rs1. This string is initialized to the last element of the token in the second index of the temporary vector. There is also a string to represent rs2. This string is initialized to the last element of the token in the third index of the temporary vector. Then there is a string to represent the immediate value. This is initialized to the token in the first index of the temporary vector. Then there are ints that use the atoi() function, from the standard c library, to convert the rs1 string, the rs2 string, and the immediate string to an int. The int storing the immediate value is anded with the hex value 0x0000FFFF, to account for sign extension. Then the int representing rs1 is shifted to the left 21 bits and the int representing rs2 is shifted to the left 16 bits. Then the opcode, which was passed in through the parameters, gets shifted to the left 26 bits and ored with the shifted rs1 int, the shifted rs2 int, and the sign extended immediate value. This is all stored into the code int, which is then converted to hex and written to the outFile using ‘0’ as a padding.

void iAlu(vector<string> type, int instr, ofstream\* outFile)

The iALU function is called in the encode function whenever the token is a math immediate type. This takes in three parameters, the vector that is temporarily holding each piece of the instruction line, an int representing the instruction opcode, and a reference to the output file. Inside the iALU function there is a string to represent rs1. This string is initialized to the last element of the token in the second index of the temporary vector. There is also a string to represent rs2. This string is initialized to the last element of the token in the first index of the temporary vector. Then there is a string to represent the immediate value. This is initialized to the token in the third index of the temporary vector. Then there are ints that use the atoi() function, from the standard c library, to convert the rs1 string, the rs2 string, and the immediate string to an int. The int storing the immediate value is anded with the hex value 0x0000FFFF, to account for sign extension. Then the int representing rs1 is shifted to the left 21 bits and the int representing rs2 is shifted to the left 16 bits. Then the opcode, which was passed in through the parameters, gets shifted to the left 26 bits and ored with the shifted rs1 int, the shifted rs2 int, and the sign extended immediate value. This is all stored into the code int, which is then converted to hex and written to the outFile using ‘0’ as a padding.

void iJump(vector<string> type, int instr, ofstream\* outFile)

The iJump function is called in the encode function whenever the token is a jump immediate type. This takes in three parameters, the vector that is temporarily holding each piece of the instruction line, an int representing the instruction opcode, and a reference to the output file. Inside the iJump function there is a string to represent rs1. This string is initialized to the last element of the token in the first index of the temporary vector. There is also a string to represent rs2. Then there is an int that uses the atoi() function, from the standard c library, to convert the rs1 string to an int. This value is then shifted to the left 21 bits. Then there is an int to represent the immediate value and an int to represent rs2. Both of these ints are initialized to zero. Then the opcode, which was passed in through the parameters, gets shifted to the left 26 bits and ored with the shifted rs1 int, the rs2 int, and the immediate value. This is all stored into the code int, which is then converted to hex and written to the outFile using ‘0’ as a padding.

void lhi(vector<string> type, int instr, ofstream\* outFile)

The lhi function is called in the encode function whenever the token is a lhi type. This takes in three parameters, the vector that is temporarily holding each piece of the instruction line, an int representing the instruction opcode, and a reference to the output file. Inside the lhi function there is a string to represent rs2. This string is initialized to the last element of the token in the first index of the temporary vector. There is also a string to represent the immediate value. This is initialized to the token in the second index of the temporary vector. Then there are ints that use the atoi() function, from the standard c library, to convert the rs2 string and the immediate string to an int. The int storing the immediate value is anded with the hex value 0x0000FFFF, to account for sign extension. Then there is an int representing rs1 which is initialized to zero. The int representing rs2 is shifted to the left 16 bits. Then the opcode, which was passed in through the parameters, gets shifted to the left 26 bits and ored with the rs1 int, the shifted rs2 int, and the sign extended immediate value. This is all stored into the code int, which is then converted to hex and written to the outFile using ‘0’ as a padding.

void rMath(vector<string> type, int instr, ofstream\* outFile)

The rMath function is called in the encode function whenever the token is an ALU R-type. This takes in three parameters, the vector that is temporarily holding each piece of the instruction line, an int representing the instruction opcode, and a reference to the output file. Inside the rMath function there is a string to represent rs1. This string is initialized to the last element of the token in the second index of the temporary vector. There is also a string to represent rs2. This string is initialized to the last element of the token in the third index of the temporary vector. Then there is a string to represent the rd value. This is initialized to the last element of the token in the first index of the temporary vector. Then there are ints that use the atoi() function, from the standard c library, to convert the rs1 string, the rs2 string, and the rd string to an int. Then the int representing rs1 is shifted to the left 21 bits, the int representing rs2 is shifted to the left 16 bits, and the int representing the rd value is shifted to the left 11 bits. Then the opcode for r-types is always zero, so this zero gets shifted to the left 26 bits and ored with the shifted rs1 int, the shifted rs2 int, and the shifted rd value. This is all stored into the code int, which is then converted to hex and written to the outFile using ‘0’ as a padding.

void branch(vector<string> type, vector<string> labels, vector<int> location, int instr, int num, ofstream\* outFile)

The branch function is called in the encode function whenever the token is a branch type instruction. This takes in six parameters, the vector that is temporarily holding each piece of the instruction line, the vector that is holding the label names, the vector that is holding the label locations, an int representing the instruction opcode, an int representing the location of the branch instruction, and a reference to the output file. Inside the branch function there is a string to represent rs1. This string is initialized to the last element of the token in the first index of the temporary vector. Then there is a string that represents the offset. This is initialized to the token in index two of the temporary vector. Next there is an int that uses the atoi() function, from the standard c library, to convert the rs1 string to an int which is then shifted to the left 21 bits. There is also and int initialized to zero representing rs2. This int is shifted to the left 16 bits. Then an int to hold the correct label location is created. The branch function then creates a loop to go through each element in the labels vector. If the element is equal to the offset string, it sets that label location int equal to the element in the location vector at the index which the element in the labels vector matches the offset. An offset int is then created which is initialized to the label location int, minus the jump location, which was passed in through the parameters, plus one. All of this is multiplied by four and the anded with 0x0000FFFF to account for sign extension. Then the opcode, which was passed in through the parameter list, is shifted over 26 bits and ored with the shifted rs1 int, the shifted rs2 int, and the offset int. This is stored into the code int, which is then converted to hex and written to the outFile using ‘0’ as padding.

void jJump(vector<string> type, vector<string> labels, vector<int> location, int instr, int num, ofstream\* outFile)

The jJump function is called in the encode function whenever the token is a jump j-type instruction. This takes in six parameters, the vector that is temporarily holding each piece of the instruction line, the vector that is holding the label names, the vector that is holding the label locations, an int representing the instruction opcode, an int representing the location of the jump instruction, and a reference to the output file. Inside the jJump function there is a string that represents the offset. This is initialized to the token in index one of the temporary vector. Then an int to hold the correct label location is created. The jJump function then creates a loop to go through each element in the labels vector. If the element is equal to the offset string, it sets that label location int equal to the element in the location vector at the index which the element in the labels vector matches the offset. An offset int is then created which is initialized to the label location int, minus the jump location, which was passed in through the parameters, plus one. All of this is multiplied by four and the anded with 0x03FFFFFF to account for sign extension. Then the opcode, which was passed in through the parameter list, is shifted over 26 bits and ored with the offset int. This is stored into the code int, which is then converted to hex and written to the outFile using ‘0’ as padding.

**Step-by-Step**

Step 1: Call main() function during the program startup

Step 2: Initialize Variables: a vector to hold the tokens (result), a vector to temporarily hold the instruction Steps (temp), a vector to hold the label names (labels), and a vector to hold the label locations (location);

Step 3: Call the readParse() function. In the parameters list pass a reference to the result vector, a reference to the labels vector, and a reference to the location vector.

Step 4: Go to the readParse() function and initialize local variables: the input file, a string (instr), an int to keep track of the location.

Step 5: While the file Steps are being read into the instr string, go to Step 6. Otherwise, go to Step 15.

Step 6: Increase index by 1 and initialize a char array with a length of 50 (str).

Step 7: Use strcpy() to transfer instr string into the str char array and initialize a char pointer (cur).

Step 8: Set cur equal to strtok(str, “ ,\n\t#()”) to parse the file whenever one of the indicated characters occurs.

Step 9 While cur does not equal null go to Step 10. Otherwise go to Step 5.

Step 10: Initialize a string(item) and call the lower() function passing in cur as a parameter.

Step 11: if the last index of the string is a colon go to Step 12. Otherwise, go to step 13.

Step 12: Save the item into a temporary string and then push it into the labels vector. Push the index into the location vector. Go to Step 14.

Step 13: Push the item string into the result vector.

Step 14: Set cur equal to strtok(NULL, “ ,\n\t#()”) to parse the next token.

Step 15: Call the encode() function. In the parameters list pass a reference to the result vector, a reference to the temp vector, a reference to the labels vector, a reference to the location vector, and a reference to the output file.

Step 16: Initialize Seven string arrays to hold the different types of instructions. Load, Store, Mathi, Branch, Mathr, Jumpi, Jump.

Step 17: Initialize Seven int arrays that hold the instruction opcodes and are parallel to the string arrays. Lopcode, sopcode, mopcode, bopcode, mropcode, jopcode, jiopcode.

Step 18: Initialize other local variables: an int to hold instruction code, and int to represent the lhi opcode, and a counter.

Step 19: Loop through each element of the result vector starting i at 0 and stopping when i equals the result vector length minus one. Perform Step 20- 40 for each iteration of the loop.

Step 20: If the token at index i of the result vector equals “lhi” pushback i, i +1, i + 2 and i + 3 into the temp vector. Otherwise go to Step 23.

Step 21: Set instruction equal to the lhicode. Call the lhi() function. Go to Step 39.

Step 22: Clear the temp vector and increase the counter by one.

Step 23: Loop through each element of the arrays. Each array has a length of 25. Start at j equals zero and end when j equals the array length minus one. Perform step 23-38 for each iteration of the loop.

Step 24: If the token at index i of the result vector equals the element in index j of the load array pushback i, i +1, i + 2 and i + 3 into the temp vector. Otherwise go to Step 27.

Step 25: Set instruction equal to the Lopcode[j]. Call the load() function. Go to Step 41.

Step 26: Clear the temp vector and increase the counter by one.

Step 27: If the token at index i of the result vector equals the element in index j of the store array pushback i, i +1, i + 2 and i + 3 into the temp vector. Set instruction equal to the sopcode[j]. Call the store() function. Go to Step 43.

Step 28: Clear the temp vector and increase the counter by one.

Step 29: If the token at index i of the result vector equals the element in index j of the mathi array pushback i, i +1, i + 2 and i + 3 into the temp vector. Set instruction equal to the mopcode[j]. Call the iAlu() function. Go to Step 45.

Step 30: Clear the temp vector and increase the counter by one.

Step 31: If the token at index i of the result vector equals the element in index j of the branch array pushback i, i +1, i + 2 and i + 3 into the temp vector. Set instruction equal to the bopcode[j]. Call the branch() function. Go to Step 47.

Step 32: Clear the temp vector and increase the counter by one.

Step 33: If the token at index i of the result vector equals the element in index j of the mathr array pushback i, i +1, i + 2 and i + 3 into the temp vector. Set instruction equal to the mropcode[j]. Call the rmath() function. Go to Step 49.

Step 34: Clear the temp vector and increase the counter by one.

Step 35: If the token at index i of the result vector equals the element in index j of the jump array pushback i, i +1, i + 2 and i + 3 into the temp vector. Set instruction equal to the jopcode[j]. Call the jJump() function. Go to Step 51.

Step 36: Clear the temp vector and increase the counter by one.

Step 37: If the token at index i of the result vector equals the element in index j of the jumpi array pushback i, i +1, i + 2 and i + 3 into the temp vector. Set instruction equal to the jiopcode[j]. Call the iJump() function. Go to Step 53.

Step 38: Clear the temp vector and increase the counter by one.

Step 39: Declare the local variables of the lhi() function: string rs2 which equals the last element of the token in the index one of the temp vector, string imm which is equal to the token in index two of the temp vector, int rs1Num = 0, int rs2Num, immNum, and code.

Step 40: Convert the final code int to hex and write it to the output file. Go to Step 22.

Step 41: Declare the local variables of the load() function: string rs1 which equals the last element of the token in index three of the temp vector, string rs2 which equals the last element of the token in index one of the temp vector, string imm which is equal to the token in index two of the temp vector, int rs1Num, int rs2Num, immNum, and code.

Step 42: Convert the final code int to hex and write it to the output file. Go to Step 26.

Step 43: Declare the local variables of the store() function: string rs1 which equals the last element of the token in index two of the temp vector, string rs2 which equals the last element of the token in index three of the temp vector, string imm which is equal to the token in index one of the temp vector, int rs1Num, int rs2Num, immNum, and code.

Step 44: Convert the final code int to hex and write it to the output file. Go to Step 28.

Step 45: Declare the local variables of the iAlu() function: string rs1 which equals the last element of the token in index two of the temp vector, string rs2 which equals the last element of the token in index one of the temp vector, string imm which is equal to the token in index three of the temp vector, int rs1Num, int rs2Num, immNum, and code.

Step 46: Convert the final code int to hex and write it to the output file. Go to Step 30.

Step 47: Declare the local variables of the branch() function: string rs1 which equals the last element of the token in index one of the temp vector, string offset which equals the element in index two of the temp vector, int rs1Num, int rs2Num, and int labeloc. Loop through the labels vector and find the label that matches the offset. Set labeloc to that iteration number of the loop. Calculate the code int.

Step 48: Convert the final code int to hex and write it to the output file. Go to Step 32.

Step 49: Declare the local variables of the rMath() function: string rs1 which equals the last element of the token in index two of the temp vector, string rs2 which equals the last element of the token in index three of the temp vector, string rd which is equal to the token in index one of the temp vector, int rs1Num, int rs2Num, rdNum, and code.

Step 50: Convert the final code int to hex and write it to the output file. Go to Step 34.

Step 51: Declare the local variables of the jJump() function: string offset which equals the element in index one of the temp vector, int labeloc. Loop through the labels vector and find the label that matches the offset. Set labeloc to that iteration number of the loop. Calculate the code int.

Step 52: Convert the final code int to hex and write it to the output file. Go to Step 36.

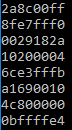
Step 53: Declare the local variables of the iJump() function: string rs1 which equals the last element of the token in index one of the temp vector, int rs1Num, int rs2Num, immNum, and code.

Step 54: Convert the final code int to hex and write it to the output file. Go to Step 38.

Step 55: Return 0

**Testing**

Input Output

SubI r12,R20,255

start: lw r7,-16(R31)

SLT r3,r1,r9

BEQZ r1, STOP

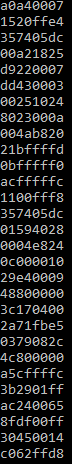
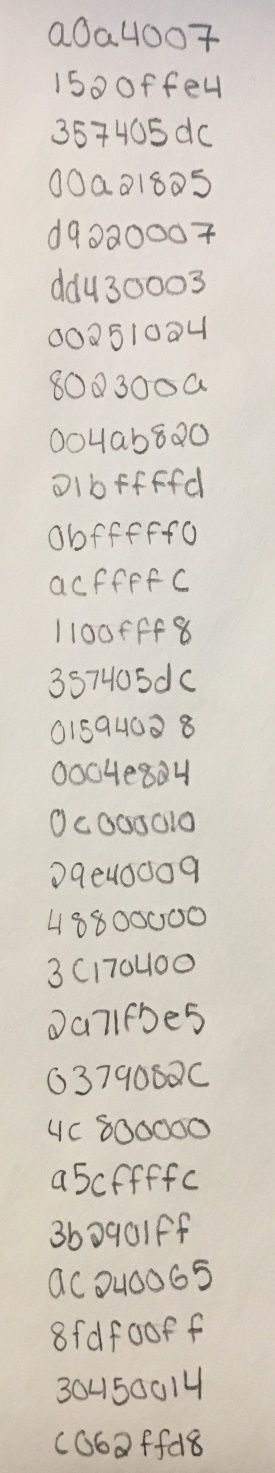
SGTI r3,r7,-5

stop: sb 16(r11),r9

jalr r4

J star

Input Output Output by hand

test1: lb r2,#7(r5)

add r3,r7,r1

addi r11,r13,-3

j test1

test2: sw -4(r7),r31

beqz r8, test2

jr r4

jalr r4

test3: subi r12,r2,13

sub r5,r9,r12

lw r3,#2(r8)

lh r7,#6(r10)0

sh #9(r3),r1

sb #7(r5),r4

bnez r9, test3

ori r20,r11,1500

or r3,r5,r2

slli r2,r9,7

srli r3,r10,3

and r2,r1,r5

test4: lb r3,#10(r1)

add r23,r2,r10

addi r31,r13,-3

j test4

test5: sw -4(r7),r31

beqz r8, test5

ori r20,r11,1500

seq r8,r10,r25

and r29,r0,r4

jal test6

subi r4,r15,9

jr r4

lhi r23,1024

subi r17,r19,-1051

test6: sle r1,r27,r25

jalr r4

sh -4(r14),r15

xori r9,r25,511

sw 101(r1),r4

lw r31,255(r30)

andi r5,r2,20

sequi r2,r3,-40

Input