**Name – Shishir Ghorashainee**

CSC 362 Homework #4

Due Date: Wednesday, October 16

Word process all answers. Show work for partial credit.

1. We have a 16Gx64 word addressable memory built from 2Gx4 RAM chips. Memory uses low-order interleave. NOTE: 16Gx64 means 16G of words, each word is 64 bits.
   1. How many RAM chips are needed to make up memory?

**Size of Memory = 16G \* 64**

**Size of Chips = 2G \* 4**

**Number of Chips to make up memory = (16G \* 64) / (2G\*4)**

**= 128 RAM Chips**

* 1. How many banks does memory have?

**Number of banks = 128 chips / 16chips per bank**

**= 8 banks**

* 1. How many bits are needed for a memory address?

**Size of memory address = Log2 (16\*2^30)**

**= 34 bits**

* 1. Of those bits from part c, which are used for the bank select, and which are used for the address in the bank?

**= Rightmost 3 bits are used for the bank select and Leftmost 31 bits are used for the address in the bank.**

For questions 2-3, provide the full MARIE RTN for these new MARIE instructions. Include the fetch, decode, get operand (if necessary) and execution stages.

1. JumpOnPos X – this is a conditional instruction which branches to address X if the result currently in the AC is positive.

**Fetch:**

**MAR 🡨 PC**

**MBR 🡨 M[MAR]**

**IR🡨MBR**

**PC 🡨 PC +1**

**Decode:**

**Decode IR[15-12]**

**Get op: // Not needed**

**Execute:**

**If (AC > 0) then PC 🡨 IR[11-0]**

1. AddIndexed X – this instruction uses indexed addressing mode to add a datum to the AC where the datum to be added is stored at memory address of X + R1 where X is part of the instruction (e.g., AddIndexed 500 uses 500 for X) and R1 is a special register in the CPU. The datum to be added to the AC is at M[X+R1] where X is given in the instruction (like 500). Assume the ALU can directly access R1 but not the MAR or IR so you will have to move the IR[11-0] address into the MBR. The result of this addition is placed back into the R1. Also assume there is a pathway from R1 to the MAR.

**Fetch:**

**MAR 🡨 PC**

**MBR 🡨 M[MAR]**

**IR🡨 MBR**

**PC 🡨 PC +1**

**Decode:**

**Decode IR[15-12]**

**Execute:**

**MBR 🡨 IR[11-0]**

**R1 🡨 R1+ MBR**

**MAR 🡨 R1**

**MBR 🡨 M[MAR]**

**AC 🡨 AC + MBR**

**R1 🡨 AC**

For 4-5, convert the C code to equivalent MARIE code. To reference numbers like 1 (for instance when doing x++), you can either use #1 or assume the variable one is storing 1.

1. x=10;

sum=0;

while(x>0) { // loop x times (count downward from y to 0 scanf(“%d”, &y); // get z from the user if(x>y) sum+=x;

else sum+=y; x--; }

printf(“%d”, sum); // output the count

**Load ten**

**Store x // 10 is stored in x**

**Clear // we use this so that we can store 0 in the Sum**

**Store Sum**

**top:**

**Load x**

**Skipcond 10 // check if (x > 0)**

**Jump done // Jump to done clause, if above condition fails**

**Input // If true, then get input from the user**

**Store y // User inputs stored in y**

**Load x**

**Subt y // Subtract y from x**

**Skipcond 10 // ( if (x-y) > 10 then skip the next line )**

**Jump else // if the condition fails, Jump to else clause**

**Load sum // Sum gets loaded if the condition is true.**

**Add x**

**Store sum // Stores value of (x + sum) from the line above**

**Load x**

**Subt one**

**Store x**

**Jump top // Jump to the top of the loop**

**else:**

**Load sum**

**Add y**

**Store sum // Stores the value after adding y to sum**

**Load x**

**Subt one // Perform (x-1) or the decrement ( x -- )**

**Store x // Once decremented, store the new value in x.**

**Jump top**

**done:**

**Load sum**

**Output**

**Halt**

1. if(x!=y&&x!=z) x++; // assume x, y, z are already loaded with values

else if(y>=z) y++;

else z++;

**Load x**

**Subt y**

**Skipcond 01 // Checks if (x = = y)**

**Jump elseif // if the above condition is fails, jump to elseif clause**

**Jump conditionTwo // if true, jump to conditionTwo clause to check x! =z**

**conditionTwo:**

**Load x**

**Subt z**

**Skipcond 01 // checks if (x = = z)**

**Jump incrementX // Jump to incrementX if x and z are not same**

**Jump elseif // (x== z) so jump to elseif clause**

**incrementX:**

**Load x**

**Add one // Add 1 to x**

**Store x // Stores the incremented x value**

**Jump done**

**elseif:**

**Load y**

**Subt z**

**Skipcond 10 // checks if (y > z)**

**Jump checkEqual // If false, then jump to conditionTwo**

**Jump incrementY // If true, then jump to incrementY clause**

**checkEqual:**

**Skipcond 01 // Checks if (y = = z) on the given 2nd line code**

**Jump else // If false then, jump to else to increment Z**

**Jump incrementY // If true then jump to incrementY**

**incrementY:**

**Load y**

**Add one**

**Store y // Stores the incremented y value**

**Jump done**

**else:**

**Load z**

**Add one**

**Store z // Stores the incremented z value**

**Jump done**

**done:**

**. . . . . . .**

1. Determine the number of cycles the following MARIE code takes to execute. Assume the input value from the user is a positive number for the first 5 inputs and 0 for the 6th input (to determine the number of times the Skipcond is true or false). Remember that we are using the RTN where a fetch takes 4 cycles, separating IR  M[MAR] into MBR  M[MAR] and IR  MBR. For skipcond, if the condition is false, the execute stage takes 1 cycle, and if the condition is true, the execute stage takes 2 cycles. Show all of your work for credit.

Clear The code is equivalent to this C code

Store Sum sum=0;

Input scanf(“%d”, &x);

Top: Skipcond 10 while(x>0) {

Jump Done sum+=x;

Add Sum scanf(“%d”, &x);

Input }

Jump Top printf(“%d”, sum);

Done: Load Sum

Output

Halt

**Clear = 6**

**Store = 8**

**Input = 6**

**Loop: 41 + 6 +40+ 30 +30 = 147 cycles**

**Skipcond: 5\*7 + 1\*6 = 41**

**Jump: 6**

**Add: 8\*5 = 40**

**Input: 6\*5 = 30**

**Jump: 6\*5 = 30**

**Done: 8 +6 +5 = 19 cycles**

**Load = 8**

**Output = 6**

**Halt = 5**

**Total = (147 + 19 + 6 + 8 + 6) cycles**

**= 186 cycles**

7. Hand-compile the following MARIE assembly program into machine code. Your machine code should be written in hexadecimal, not binary. The first instruction in the program is at memory address 889. The variables (A, I, One, Ten) are stored in the memory locations immediately after Halt in order (A, I, One, Ten). Remember that addresses are in hexadecimal, not decimal.

Clear

Store A

Store I

Top: Subt Ten

Skipcond 01

Jump Done

Input

Add A

Store A Load I

Add One

Store I

Jump Top

Done: Load A

Output

Halt

// variables stored here in order A, I, One, Ten

**A is stored at 899**

**I is stored at 89A**

**One is stored at 89B**

**Ten is stored at 89C**

|  |  |  |
| --- | --- | --- |
| **889** | **Clear** | **A000** |
| **88A** | **Store A** | **2899** |
| **88B** | **Store I** | **289A** |
| **88C** | **Top: Subt Ten** | **489C** |
| **88D** | **Skipcond 01** | **8400** |
| **88E** | **Jump Done** | **9896** |
| **88F** | **Input** | **5000** |
| **890** | **Add A** | **3899** |
| **891** | **Store A** | **2899** |
| **892** | **Load I** | **189A** |
| **893** | **Add One** | **389B** |
| **894** | **Store I** | **289A** |
| **895** | **Jump Top** | **988C** |
| **896** | **Done: Load A** | **1899** |
| **897** | **Output** | **6000** |
| **898** | **Halt** | **7000** |