CSE 141L
Lab 1
Zhisheng Liu A123991234
Amal Alhaidari A98069538

Components of lab 1:

1. (10 pts) Introduction.
   This should include the name of your architecture, overall philosophy, specific goals strived for and achieved.

   Name of the architecture: WDIGMI - What Did I Get Myself Into

   Overall philosophy:
   We chose an accumulator type ISA because it provides a balance between instruction bits for opcodes and registers. Our philosophy was to spend a lot of time on the algorithms that solve the 3 presented problems, translate them into an ISA type of code, and then go back and adjust the algorithms to fit an accumulator type ISA.

   Goals: design an ISA for a special purpose processor that can solve the 3 presented problems.

2. (5 pts) Instruction formats.
   Give all formats and an example of each.

   We have 2 types of instructions: Normal-Type Instruction and Special-Type Instruction. The type is determined by the MSB-bit of the 9-bit instruction. We call the first bit a "type bit." The type bit determines how the next 8 bits in the instruction are interpreted.

   Normal-Type Instruction:
   Here, the type bit is 0. The rest of the 8 bits are as follows: 4 bits are assigned for the opcode (16 opcodes), and 4 bits are assigned for the operand (16 registers). Example: <u>0</u> 0000 0001, take $r1. The underlined bit is the type bit. It indicates that this instruction is a Normal-Type Instruction, and that the next 4 bits are opcodes: 0000 is the "take" opcode. The last 4 LSB indicate a register. In this case $r1.

   Special-Type Instruction:
   Here, the type bit is 1. The remaining 8 bits are all used to indicate an immediate (0-255). This type is designed to store the immediate into the accumulator register $r0, thus only one operation exists in Special-Type Instructions. Example: <u>1</u> 0000 0001, takei #1. The

underlined bit is the type bit. It indicates that this instruction is a Special-Type Instruction, and that the next 8 bits are the immediate: 0000 0001 or 1. Therefore, the value 1 is stored into the accumulator $r0.

3. (10 pts) Operations.
Give all instructions supported and their opcodes/formats.
Instruction format
NOTE:

- for the encode part, 'i' represent immediate type, 'r' represent register type, 'x' represent don't care or undefined
- opcodes are labeled with x in normal type: 0 x x x x _ _ _ _
- General purpose registers are indicated with a dollar sign ($), Immediate is indicated with a pound sign (#)
- r16 is the extra 1 bit register for overflow detection, which does NOT count as one of the 16 registers(r0~r15)

0. takei -- take immediate [special I-type instruction]
Description: an immediate is move(take) into the accumulator register
operation: $accumulator = #imm
syntax: takei #imm
type: special-type
encode: 1 iiii iiii

1. take
description: the content of the operand register is move(take) into the accumulator register
operation: $accumulator = $reg
syntax: take $reg
type: normal-type
encode: 0 0000 rrrr

2. put
description: the content of the accumulator register is move(put) into the operand register
operation: $reg = $accumulator

syntax: put $reg
type: normal-type
encode: 0 0001 rrrr

3. load
description: a byte is loaded into the accumulator from the address specified by operand register
operation: $accumulator = MEM[$reg]
syntax: load $reg
type: normal-type
encode: 0 0010 rrrr

4. store
description: the content of accumulator is stored at the address specified by operand register
operation: MEM[$reg] = $accumulator
syntax: store $reg
type: normal-type
type: R-type
encode: 0 0011 rrrr

5. xor -- bitwise exclusive or
description: exclusive or the operand register with accumulator and store the result into accumulator
operation: $accumulator = $accumulator ^ $reg
syntax: xor $reg
type: normal-type
encode: 0 0100 rrrr

6. nand -- nand gate
description: combination of and and not operation
operation: $accumulator = $accumulator nand $reg
syntax: nand $reg
type: normal-type
encode: 0 0101 rrrr

7. shl -- shift left
description: Shifts accumulator left by the amount specified by a register. Zeroes are shifted in.

operation: $accumulator = $accumulator << $reg
syntax: shl $reg
type: normal-type
encode: 0 0110 rrrr

8. shr -- shift right
description: Shifts accumulator right by the amount specified by a register. Zeroes are shifted in.
operation: $accumulator = $accumulator >> $reg
syntax: shl $reg
type: normal-type
encode: 0 0111 rrrr

9. b0 -- branch on true
description: branch to the branch register = &[LABLE] if the accumulator has a value of 0
operation: if $accumulator == 0, move program counter to &[LABLE]
syntax: b1 LABLE
type: normal-type
encode: 0 1000 rrrr

10. lsn -- check if acc is less than register
description: compare a register with the accumulator and put 1 in accumulator if accumulator is less than register
operation: if $accumulator < $reg, $accumulator = 1, else accumulator = 0
syntax: lsn $reg
type: normal-type
encode: 0 1001 rrrr

11. eql
description: check if acc is equal to register
operation: if $accumulator == $reg, $accumulator = 1, else $accumulator = 0
syntax: eql $reg
type: normal-type
encode: 0 1010 rrrr

12. add
description: add the operand register to the accumulator
operation: $accumulator = $accumulator + $reg
syntax: add $reg

type: normal-type
encode: 0 1011 rrrr

13. sub
description: sub the operand register from the accumulator
operation: $accumulator = $accumulator - $reg
syntax: sub $reg
type: normal-type
encode: 0 1100 rrrr

14. of0 -- set overflow bit register to zero
description: set overflow bit register to zero
operation: $r16 = 0
syntax: of0
type: normal-type
encode: 0 1101 xxxx

15. halt
description: halt the machine
operation: halt the machine
syntax: halt
type: normal-type
encode: 0 1110 xxxx

16. tba -- to be defined
description: to be defined
operation: to be defined
syntax: to be defined
type: normal-type
encode: 0 1111 xxxx

4. (5 pts) Internal operands.
How many registers are supported? Is there anything special about the registers?
16 register are supported, $r0~$r15, $r0 is the accumulator register. We also use an extra $r16 as a 1-bit register to detect overflow.

5. (5 pts) Control flow (branches).
What types of branches are supported? How are the target addresses calculated? What is the maximum branch distance supported?

Our only branch instruction is indicated by the b0 opcode: branch when the accumulator is equal to 0. We support relative addressing when calculating the address to branch to. This value is the address from the branch instruction with an offset in the range of -128~127. So our maximum branch distance is -128~127 away from the branch instruction's address. The target address of the label is stored in a register.

6. (5 pts) Addressing modes.
   What memory addressing modes are supported? How are addresses calculated? Give examples.
   We adopted register indirect and some relative addressing mode in our design.
   For example, we load specific content from the memory by instruction like:
   ```
   load  $register
   ```
   For sequential address after a starting point, we do relative addressing by adding an offset to it. For example, while we are loading 64 bytes starting from address 32, what we did is set up a counter register starting with 0, and add it to 32 as an offset and load their sum as the memory address:

7. (5 pts) How large is the main memory?

256 byte. since we use register indirect addressing, and each register is 8 bit wide, which map to 2^8 addresses.

8. (10 pts) In what ways did you optimize for dynamic instruction count? Please explain.

   - simplified the condition statements before we converted our c code into assembly
   - short circuited loops by exiting early if a certain condition is met: problem #19 when hamming distance 8 is reached we stop checking other elements in the array and end the program.
   - eliminate duplicate accumulator assignment: if the value in the accumulator happens to be the same as the value we are about to assign to it, then we can use the preexisting accumulator value rather than redundantly reassign the preexisting value to the accumulator
   - exploit the accumulator as much as possible: once a value goes into the accumulator, we use that value to do all relevant work before we update the accumulator

9. (10 pts) In what ways did you optimize for ease of design? Please explain.

We chose an accumulator type ISA because this way we would save extra instruction bits since the accumulator register is always implied as the destination register in the instruction rather than have a source and destination registers take up space in the instruction.

For logic and control flow, comparative operations are used right before a branch operation to a different instruction takes place. The boolean result of the comparison is stored in the accumulator, 1 for true and 0 for false, and can be followed by a branch instruction to control the flow of the program.

For the special purpose of the given three program, we design a special-type instruction by making the MSB of the 9 bit instruction as the "type bit", and when it is one, the machine will forget opcode and treat the rest 8 bit as an immediate. This design will lead to a quick access of an immediate range from -128~127, it is not a large range but it is pretty sufficient for the three programs given. For example, the largest loop counter we encounter is 64 bytes long array in problem 18, which is manageable by our design.

10. (10 pts) In what ways did you optimize for short cycle time? Please explain.

make instruction as primitive as possible, and we use accumulator type instruction which is more lower level, for example, we only have one branching instruction

```
b0 LABLE
```

it only branch to the label if the accumulator==0, unlike complex multi-operand instructions such as beq and bne, it takes less cycle times but we can still achieve functions like beq and bne by manipulating it.

11. (5 pts) If you optimized for anything else, what and how? (It's OK if you didn't)

Our Special-Type Instruction design allows us to use immediates that are encoded as part of the instructions. By approaching it this way, we minimized the store and load operations done on data memory by opting out on using a lookup table. Instead, our special type instruction fetches immediates from the instruction memory.

12. (5 pts) Your chief competitor just announced a load-store ISA with two explicit operands (one source register same as destination), four registers (i.e., a 2-bit register specifier), and

16 instructions (4 opcode bits). Tell us why your ISA is better, and give examples to support your claim.

We have 12 more registers which allows the person using our ISA to have an easier time translating a higher level code into its assembly counterpart. As opposed to spending a lot of time trying to figure out how to use the registers as efficiently as possible.

Another outcome of having more registers is that there will be less access (loads and stores) to data memory. Less access to data memory reduces the runtime of the program since data stored in registers which are faster to operate on.

13. (5 pts) What do you think will be the biggest general-purpose weakness in your design? That is, what don't you have that you will miss the most if you were to have to write other, longer, programs?

Out instruction count would be high for longer programs because:

1) we use an accumulator type ISA so we always need to store the value of the accumulator elsewhere before operating on the accumulator again.

2) we use low level instructions such as nand for NAND to create bitwise-AND, and bitwise-NOT. This will slow down longer programs since they will require a lot of instructions.

14. (a). (5 pts) What would you have done differently if you had two more bits for instructions? Please explain.

we would assign one more bit to my "type bit," that way we will have (2^2) 4 special instruction, in which we can categorize our branch instruction as the special instructions, which has 9 bit for immediate address, that way we don't have to store address into register for branching purpose.

the other bit will inevitably go into op code so that we have more convenient instructions to reduce our dynamic instruction count.

(b). (5 pts) Two fewer bits? Please explain.

Reduce the bits representing the registers from 4 to 3: find a simpler algorithm for the programs which requires less variables. This will shrink the size of the register file. That way we will have 3 bits (originally 4 bits) for registers, which means we can have up to 8 registers.

And then we remove the other bit by getting rid of the “type bit” (MSB defined in a previous part in the report), and use a lookup table for large immediate storage.

Or potentially use less opcodes depending on the complexity/simplicity of the program and its interpretation into an ISA.

15. (5 pts) Can you classify your machine in any of the classical ways (e.g., stack machine, accumulator, register, load-store)? If so, which? If not, come up with a name for your class of machine.

Our machine can be classified as mixed version of accumulator, since it adopts accumulator type operation while it take advantages of other type such as reg-mem, reg-reg.

16. (5 pts) Give an example of an “assembly language” instruction in your machine, then translate it into machine code.

instruction example: put r10

operation: the content of the accumulator register is move(put) into the register r10

machine code: 0 0001 1010

Three problem assembly bellow:
c code is post on https://github.com/zhl108/CSE-141L, the repo is currently private, you can send me an email at(zhl108@ucsd.edu) with an github username so that I can add you as collaborator to view them

---

17. The total dynamic count for part b is 631
assembly code:
```
/*
r0 = accumulator
r1 = a1
r2 = a2
r3 = b1
r4 = b2
```

```
r5 = s1
r6 = s2
r7 = s3
r8 = s4
r9 = upper bound 8  /sign_a  /temp
r10= loop counter i /sign_b
r11= isNeg         /
r12= sum_high
r13= sum_low
r14= partial_high
r15= partial_low
r16 = overlow_bit (special 1 bit register)
*/

   //load operand a1,a2,b1,b2
   takei #1
   load r0
   put r1     //r1=a1

   takei #2
   load r0
   put r2     //r2=a2

   takei #3
   load r0
   put r3     //r3=b1

   takei #4
   load r0
   put r4     //r4=b2

   //declare sign_a, sign_b
   takei #7
   put r12    //use r12(sum_high) to temp record shift amount
   take r1
   shr r12
   put r9     //sign_a = r9

   take r3
```

```
    shr r12
    put r10     //sign_b = r10

    takei #0
    put r11         //isNeg=0
    takei #1        //acc=1
    eql r9          //check if r9==1, if yes, acc=1
    b0 S_1
    takei #0
    eql r10
    b0 S_1
    takei #1
    put r11         //isNeg=1
S_1:
    takei #0
    eql r11
    b0 S_2
    takei #0
    eql r9
    b0 S_2
    takei #1
    b0 S_2
    put r11         //implicitly have r11(isNeg)=1
S_2:

    //do 2's complement on A and B
    //r1=a1, r2=a2, r3=b1, r4=b2
    //check if r9(sign_a)==1, if yes, a1<0
    takei #1
    eql r9
    b0 S_3
    take r1
    nand r0
    put r1
    take r2
    nand r0
    put r2
    take #-1
    b0 S_4
```

```
   takei #0       //a2=0;
   put r2        //a1=a1+1;
   takei #1
   add r1
   put r1
   takei #0       //jump over the else case
   b0 S_3
S_4:
   takei #1       //a2=a2+1
   add r2
   put r2
S_3:

   //r1=a1, r2=a2, r3=b1, r4=b2
   //check if r10(sign_b)==1, if yes, b1<0
   takei #1
   eql r10
   b0 S_3
   take r3
   nand r0
   put r3
   take r4
   nand r0
   put r4
   takei #-1
   b0 S_6
   takei #0       //b2=0; (r4)
   put r4        //b1=b1+1;
   takei #1
   add r3
   put r3
   takei #0       //jump over the else case
   b0 S_5
S_6:
   takei #1       //b2=b2+1
   add r4
   put r4
S_5:
   //set s1~s4=0
   takei #0
   put r5
   put r6
   put r7
   put r8

   //a2xb2 (r2xr4)--------------------------------
   //reset partial sum to zero
   put r12
   put r13
   put r14
   put r15

   put r10        //i=0
A2B2:

   //mask=1 << i
   takei #1
   shl r10     //acc = mask
            //acc and b2(r4)
   nand r4
   nand r0     //create an and gate: acc = b_bit = b2 & mask

   b0 A2B2_S
   take r2
   shl r10
   put r15     //save partial_low

   takei #8
   sub r10     //8-i
   put r9
   take r2
   shr r9      //a2>>8-i
   put r14     //save partial_high

   of0        //TODO: new instruction - set overlow_bit to zero
   take r13
   add r15
   put r13     //sum_low = sum_low + partial_low

   take r12
   add r14
   add r16     //add overflow bit register
   put r12
A2B2_S:
   //check counter at the end of loop
   takei #1
   add r10
   put r10
   takei #8
   put r9          //r9 is the upper limit of loop
   take r10
   lsn r9          //check if i<8
   b0 A2B2

   //s3 = s3 + sum_high = r7 + r12
   //s4 = s4 + sum_low = r8 + r13
   take r7
   add r12
   put r12
   take r8
   add r13
   put r13

   //a1xb2 (r1xr4)--------------------------------
   //reset partial sum to zero
   take #0
   put r12
   put r13
   put r14
   put r15

   put r10        //i=0

A1B2:
   //mask=1 << i
   takei #1
   shl r10
            //acc and b2(r4)
   nand r4
   nand r0    //create an and gate: acc = b_bit = b2 & mask

   b0 A1B2_S
   take r1
   shl r10
   put r15    //save partial_low

   takei #8
   sub r10    //8-i
   put r9
   take r1
   shr r9     //a1>>8-i
   put r14    //save partial_high

   of0        //set overlow_bit to zero
   take r13
   add r15
   put r13    //sum_low = sum_low + partial_low

   take r12
   add r14
   add r16    //add overflow bit register
   put r12
A1B2_S:
   //check counter at the end of loop
   takei #1
   add r10
   put r10
   takei #8
   put r9         //r9 is the upper limit of loop
   take r10
   lsn r9
   b0 A1B2

   //s2(r6) = s2 + sum_high = r6 + r12 (of)
   //s3(r7) = s3 + sum_low = r7 + r13 (of)
   of0
   take r7
   add r13
   put r13
   take r16
   b0 A1B2_OF_1    //if(s3_overflow)?
   of0
   takei #1
   add r6
   put r6          //if of, s2+=1
   take r16
   b0 A1B2_OF_2    //if adding one to s2 cause of?
   of0
   takei #1
   add r5
   put r5          //s1 = s1 + 1

A1B2_OF_2:
A1B2_OF_1:

   of0
   take r6
   add r12
   put r12
   take r16
   b0 A1B2_OF_3
   takei #1
   add r5
   put r5          //s1 += 1

A1B2_OF_3:

   //a2xb1 (r2xr3)--------------------------------
   //reset partial sum to zero
   take #0
   put r12
   put r13
   put r14
   put r15
   put r10        //i=0
A2B1:

   //mask=1 << i
   takei #1
   shl r10
              //acc and b1(r3)
   nand r3
   nand r0     //create an and gate: acc = b_bit = b1 & mask

   b0 A2B1_S
   take r2
   shl r10
   put r15     //save partial_low

   takei #8
   sub r10     //8-i
   put r9
   take r2
   shr r9      //a2>>8-i
   put r14     //save partial_high

   of0        //set overlow_bit to zero
   take r13
   add r15
   put r13     //sum_low = sum_low + partial_low

   take r12
   add r14
   add r16     //add overflow bit register
   put r12
A2B1_S:
   //check counter at the end of loop
   takei #1
   add r10
   put r10
   takei #8
   put r9         //r9 is the upper limit of loop
```

```
    take r10
    lsn r9
    b0 A2B1

    //s2(r6) = s2 + sum_high = r6 + r12 (of)
    //s3(r7) = s3 + sum_low = r7 + r13 (of)
    of0
    take r7
    add r13
    put r13
    take r16
    b0 A2B1_OF_1    //if(s3_overflow)?
    of0
    takei #1
    add r6
    put r6          //if of, s2+=1
    take r16
    b0 A2B1_OF_2    //if adding one to s2 cause of?
    of0
    takei #1
    add r5
    put r5          //s1 = s1 + 1

A2B1_OF_2:
A2B1_OF_1:

    of0
    take r6
    add r12
    put r12
    take r16
    b0 A2B1_OF_3
    takei #1
    add r5
    put r5          //s1 += 1

A2B1_OF_3:

//a1xb1 (r1xr3)--------------------------------
```

```
   //reset partial sum to zero
   take #0
   put r12
   put r13
   put r14
   put r15

   put r10        //i=0
A1B1:

   //mask=1 << i
   takei #1
   shl r10
            //acc and b1(r3)
   nand r3
   nand r0     //create an and gate: acc = b_bit = b1 & mask

   b0 A1B1_S
   take r1
   shl r10
   put r15     //save partial_low

   takei #8
   sub r10     //8-i
   put r9
   take r1
   shr r9      //a1>>8-i
   put r14     //save partial_high

   of0        //set overlow_bit to zero
   take r13
   add r15
   put r13     //sum_low = sum_low + partial_low

   take r12
   add r14
   add r16     //add overflow bit register
   put r12
A1B1_S:
   //check counter at the end of loop
   takei #1
   add r10
   put r10
   takei #8
   put r9          //r9 is the upper limit of loop
   take r10
   lsn r9
   b0 A1B1

   //s1(r5) = s1 + sum_high = r5 + r12
   //s2(r6) = s2 + sum_low= r6 + r13 (of)
   of0
   take r6
   add r13
   put r6
   take r16
   b0 A1B1_OF_1
   take #1
   add r5
   put r5
A1B1_OF_1:
   take r5
   add r12
   put r5


   //if isNeg is true, 2's complement result

   take r11
   b0 SKIP
   take r5
   nand r0     //nand with acc will create not gate
   put r5      //s1 = ~s1
   take r6
   nand r0
   put r6
   take r7
   nand r0
```

```
    put r7
    take r8
    nand r0
    put r8

    of0
    takei #1
    add r8
    put r8
    take r16
    b0 SKIP        //check if s4+1 is overflow
    of0
    takei #1
    add r7
    put r7
    take r16
    b0 SKIP
    of0
    takei #1
    add r6
    put r6
    take r16
    b0 SKIP
    takei #1
    add r5
    put r5
SKIP:
    //write the result s1~s4 into memory location 5~8
    takei #5
    store r5
    takei #6
    store r6
    takei #7
    store r7
    takei #8
    store r8
END
```

18. The total dynamic count for part b is 12539

assembly code:

```
/*
r0 = accumulator
r1 ~ r5 = single_match ~ quintuple_match
r6 = string / temporary
r7 = temp_string
r8 = key
r9 = isOne
r10= result
r11= count
r12= counter of LOAD_MEM
r13= counter of FIVE
r14= memory address
r15= temporary shift amount
*/
    //initialize count and match counter
    takei #0

    put r1          //single_match=0
    put r2          //...
    put r3          //
    put r4          //
    put r5          //quintuple_match=0

    //start loading key and string from memory
    takei #9         //take 9 into acc
    put r14          //r14=9
    load r14          //load the byte of memory at address r14 (which is 9) into accumulator

    put r8          //put the content of accumulator into r8, key stored

    //set up r12, counter of the loop LOAD_MEM
    takei #0         //load immediate 0 into accumulator
    put r12          //put 0 into r12, r12 is the counter to loop through 64 bytes

LOAD_MEM:
    takei #0
    put r11          //count=0, reset the count at begin of loop loading string
```

```
	takei #32          //take address 9 into acc
	add r12            //add the address offset into accumulator
	put r14            //r14=32+r13
	load r14           //load the byte of memory at address 32+r12 into accumulator

	put r6             //save the string you are comparing into r6

	//key = key << 4;
	//key = key >> 4;
	takei #4
	put r15            //r15 = shift amount = 4
	take r8            //take key into accumulator, acc = key
	shl r15            //shift the key to the left by 4 bit
	shr r15            //shift the key to the right by 4 bit
	put r8             //put key back

	//set up counter of loop FIVE r13
	takei #0
	put r13

FIVE:
	//temp_string(r7) = string(r6)
	take r6
	put r7

	take r7     //acc=r7=temp_string
	shl r13     //temp_string << i
	shr r13     //temp_string >> i
	put r7      //save temp_string

	//acc = 4-i
	takei #4
	sub r13
	put r6      //since we do NOT use r6 after we load the string from memory, reuse it

	//temp_string = temp_string << (4-i)
	take r7     //acc = temp_string
	shl r6      //shift left (4-i)
```

```
        put r7      //store the result back to temp_string

        //result = temp_string ^ key
        xor r8
        put r10

        //if(!result) count++
        takei #0
        eql r10            //if result == 0, count++ acc=1
        b0 SKIP

        takei #1
        add r11
        put r11             //count++
SKIP:
        //r13(i)++, if i<5 loop again
        takei #5
        put r15
        takei #1
        add r13
        put r13
        lsn r15
        b0 OUT_FIVE
        takei #0
        b0 FIVE

OUT_FIVE:
        //check count and increment corresponding match
        takei #1
        eql r11
        put r15
        takei #0
        eql r15
        b0 SINGLE

        takei #2
        eql r11
        put r15
        takei #0
```

```
    eql r15
    b0 DOUBLE

    takei #3
    eql r11
    put r15
    takei #0
    eql r15
    b0 TRIPLE

    takei #4
    eql r11
    put r15
    takei #0
    eql r15
    b0 QUADRUPLE

    takei #5
    eql r11
    put r15
    takei #0
    eql r15
    b0 QUINTUPLE

SINGLE:
    takei #1
    add r1
    put r1
    takei #0
    b0 BREAK
DOUBLE:
    takei #1
    add r2
    put r2
    takei #0
    b0 BREAK
TRIPLE:
    takei #1
    add r3
```

```
	put r3
	takei #0
	b0 BREAK
QUADRUPLE:
	takei #1
	add r4
	put r4
	takei #0
	b0 BREAK
QUINTUPLE:
	takei #1
	add r5
	put r5
BREAK:
	//r12++, if i<64 loop LOAD_MEM
	takei #64		//acc = 64
	put r15
	takei #1
	add r12
	put r12
	lsn r15
	b0 OUT_LOAD_MEM

	takei #0
	b0 LOAD_MEM

OUT_LOAD_MEM:
	//store match# into memory
	takei #10
	put r6
	take r1
	store r6   //mem[r6] = mem[10] = r1 = single_match

	takei #11
	put r6
	take r2
	store r6   //mem[r6] = mem[11] = r2 = double_match

	takei #12
```

```
        put r6
        take r3
        store r6    //mem[r6] = mem[12] = r3 = triple_match

        takei #13
        put r6
        take r4
        store r6    //mem[r6] = mem[13] = r4 = quadruple_match

        takei #14
        put r6
        take r5
        store r6    //mem[r6] = mem[14] = r5 = quintuple_match
    END
```

19. The total dynamic count for part b is 28888

assembly code:

```
takei #127              //acc = 127 mem&
put $r11                //returnReg = 127 mem&
takei #1
add $r11                //acc = 128 mem&
put $r5                 //i = r5 = acc = 128 mem&
takei #2
add $r11                //acc = 129 mem&
put $r6                 //j = r6 = acc
takei #0
put $r2                 //r2 = acc = biggestHamDist
put $r4                 //r4 = byte = acc = 0

Binomial1:                          //for loop from i=0 to 18 (19 times not 20)

        Binomial2:                  //for loop from j=i+1 to 19
                load $r5            //acc = mem[i]
                put $r7             //r7 = acc = mem[i]
                load $r6            //acc = mem[j]
                put $r8             //r8 = acc = mem[j]
                take $r7            //acc = mem[i]
                xor $r8             //acc = mem[i] ^ mem[j]
                put $r9             //dist = acc = mem[i] ^ mem[j]
                takei #0
                put $r3             //currHamDist = acc = 0

                CheckLSB:           //for loop from byte = 0 to 7 (8bits)
                        takei #1            //mask
                        nand $r9            //acc = !(dist & mask)
                        nand $r0            //acc = !(acc & acc) = dist & mask
                        put $r10            //temp r10 = dist & mask
                        takei #1
                        eql $r10            //if acc == temp, acc = 1
                        b0 NoMatch          //branch if acc == 0, acc != temp
                        takei #1
                        add $r3             //acc = r3 + 1 = currHamDist + 1
                        put $r3             //currHamDist++
                        take $r2            //acc = r2 = biggestHamDist
                        lsn $r3             //if acc=r2 < r3, if biggestHamDist < currHamDist
                        b0 NoMatch          //branch if !(biggestHamDist < currHamDist)
                        take $r3            //acc = r3 = currHamDist
                        put $r2             //r2 = acc = r3, biggestHamDist = currHamDist
        takei #8
        eql $r2                 //if acc == r2, if biggestHamDist == 8, acc = 1
        b0 NoMatch
        takei #0
        b0 ReturnResult

        NoMatch:                //here if my if statement checks fail
        takei #1
        put $r10                //temp = r10 = 1
        take $r9                //acc = r9 = dist
        shr2 $r10               //acc = r9 >> 1 = dist >> 1
        takei #1
        add $r4                 //acc = byte+1
        put $r4                 //r4 = byte = acc = byte+1, byte++
        takei #8
        put $r10                //temp = 8
        take $r4                //acc = r4 = byte
        lsn $r10                //if acc < 8, acc = 1
        put $r10                //temp = acc
        takei #0
        eql $r10                //if acc == temp, temp == 0, acc = 1
        b0 CheckLSB             //byte < 8, acc = 0
takei #1
add $r11                //acc = 1+127
put $r10                //temp = 128
take $r5                //acc = i
sub $r10                //acc = i - 128
add $r6                 //acc = i - 128 + j
put $r10                //temp = i - 128 + j
takei #1
add $r10                //acc = i-128+j+1
put $r6                 //j = r6 = acc = i-128+j+1, c code: j=j+i+1
takei #2
add $r11                //acc = 2+127
put $r10                //temp = 129
take $r6                //acc = j
sub $r10                //acc = j-129
put $r10                //temp = j-129
takei #20
put $r7                 //temp2 = 20
take $r10               //acc = temp = j-129
lsn $r7                 //if acc < temp2=20, acc = 1
put $r10                //temp = acc
```

```
                takei #0
                eql $r10                //if acc == temp, temp == 0, !(j<20), acc = 1
                b0 Binomial2            //(j < 20)

        takei #1
        add $r5                 //acc = i + 1
        put $r5                 //i++
        takei #128
        put $r10                //temp = 128
        take $r5                //acc = i
        sub $r10                //acc = i - temp = i - 128
        put $r10
        takei #19
        put $r7                 //temp2 = 19
        take $r10               //acc= temp = i-128
        lsn $r7                 //if acc < 19, acc = 1
        put $r10                //temp = acc
        takei #0
        eql $r10                //if acc == temp, temp == 0, !(j<20), acc = 1
        b0 Binomial1            //branch to Binomial1 if i < 19
ReturnResult:
take $r2                //acc = r2 = biggestHamDist
store $r11              //mem[127] = acc = biggestHamDist
```