Report CAO

Team 44

Marleen Siebons (S2504588), Elise Korsmit (S2999471), Guusje van Leersum (S3164918)

Contents

Task 1: Hamming Distance calculator in C

While writing the code for task 1, the first thing we did was carefully consider the trade-off between memory utilization and execution speed. No limits were imposed on the value of L (the length of the strings, but the number of strings given by the users is expected to be larger than 1.

A lot of effort was put into achieving a program capable of some dynamic memory allocation. The execution speed was not our main concern, but it was not completely overlooked either.

The function rand already generates a random number, which inside of the computer is a random sequence of bits, which we use. You can run “sh Task1Script.sh” in the command line, when Task1Script.sh shows if you type ls, to run the code for the specified numbers in the assignment. To run the code for N bitstrings, of length L to compare, with seed S, you simply run:

```
cc -o Task1FromScript Task1Bitstring.c
./Task1FromScript N L S
```

from the command line.

Task 2: Assembly implementation

Suitable data representation in memory: We save the bitstring one bit (as a character in ASCII) per byte in memory. This organically evolved from how we polled the user input, where we read the 1’s and 0’s as text, hence we use ASCII characters. We store each consecutive ASCII character in consecutive bytes, where different bitstrings are aligned on words (4bytes). This is not at all an efficient use of memory, but it works.

Because it is easy (and intended) for users to configure the values for L and N, we do not allow the user to put in any bitstrings that are shorter than the required amount (throws no errors, assumes competent users). If the user puts in a string that is longer, then it is just capped off. This was specified in the assignment, and the length and number of strings can be modified at the top of the code.

A grand total of 6 instructions and 3 instruction types were used in the disassembled HD calculation. If one looks at our HD calculator implementation with our homemade ISA (task3) in mind, one could say that we only used 6 instructions, of 2 instruction types.

Task 3: a Better ISA for our Hamming Distance calculator: BMIPS

1. Word size and data types
    a. Word size: 24-bit
    b. Data types: Since both signed and unsigned integers are used, we will include both. Both characters and addresses are used in the HD calculator. Floats will not be in the BMIPS ISA as they are of no importance to our implementation.
2. Register set
    a. General-purpose registers (GPR): the HD calculator uses 6 saved registers and 7 temporary registers.
    b. Special-purpose registers (SPR): the program uses a Program Counter (PC). The SP,GP, FP, RA, K0, and K1 registers keep their values as before the program started, probably due to the software used during our implementation. The registers we use are shown in table 3

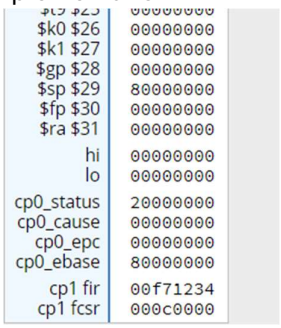

Figure 1: Special registers used with MIPS assembler

3. Instruction Format
    a. This ISA (BMIPS ISA) uses an RISC (Reduced Instruction Set Computer); Simple instructions that execute in one or a few instructions. The number of instructions has been reduced due to the restriction to calculate Hamming Distances (HD).
    b. Opcode: These are used to specify the operation and BMIPS' opcodes use less bits than MIPS' opcodes as there are less operations.
    c. Operands: add, sub, mul, addi, lb, beq, which we used to defined pseudo instructions: nop, li, j. See tables 4 and 5.
    d. Instruction length: this ISA is of RISC type and therefore the instruction length will be fixed. The size of 24 bits (3bytes) was chosen based on the analysis included in Appendix A.
    e. Common Instruction formats:
        i. R-Type (Register-based): Operations like add, sub, etc.
        ii. I-Type (Immediate-based): Instructions involving immediate values. These instruction types are in the MIPS ISA and used in our program. However, in the BMIPS ISA, the J-type instructions are redundant as they can be implemented differently.

4. Instruction Set Design
    a. Later on in this document you can find a table of all instructions used in the HD calculator, with reduced opcodes, which we constructed based on the regular MIPS ISA. We took this as a basis for our BMIPS, for which we also included tables.
    Due to the smaller number of instructions used in our code, the entropy of the operands is much lower and less bits can be used to describe them. This would make the execution faster and more efficient as the system would not have to check for as many instructions and the smaller opcodes would also mean that the program can take up less space.

5. Memory Addressing Modes
    a. Immediate
    b. Registers
    c. Direct

6. Addressing the Stack
    a. No PUSH or POP operations are used in our HD calculator, and therefore this ISA will not include any operations for explicit interaction with the stack.

7. Instruction Execution Model
    a. Pipeline: The five stages of pipelining in MIPS are Instruction Fetch, Instruction Decode, Instruction executing, memory read, and write back results to memory. Modern pipelined processors tend to have more pipelined stages than this to make them more efficient. Our HD calculator does not use of delayed branches. This means that prefetching the targets of the branches could make large improvements in the speed of the execution of our program, since the program has to wait until the decision for the branch has been made.
    Another improvement that could be made to the MIPS pipelines is an Instruction prefetch pipeline. Since Instruction execution takes more time (on average) than fetching, we can fetch multiple instructions at once to make better and more efficient use of the executing pipeline. One potential hazard is that branches would make prefetched instructions useless and maybe even problematic, so that would have to be taken into consideration.
    b. Out-of-Order execution: This can be used to enhance the performance of our program. A lot of undelayed branches are used, so implementing this optimization technique well will lead to notable (positive) results. This can be done by predicting outcomes of branches, if we delay our branches.

8. Exceptions and Interrupt Handling
    a. As long as a user's input can be translated to 0's and 1's by the computer, it should not cause issues in our program. Because this ISA is for only one short program there is not a lot of room for interrupts or exceptions to appear and no further definitions are necessary.

9. Performance and Power Considerations
    a. Due to our RISC architecture, this ISA will be more power-efficient (simpler instructions). However, this does mean that a larger number of instructions gets used, so the performance of the program could be lacking when compared to another program with similar goals but no strict requirements on the number of instructions.

10. Backwards Compatibility
    a. This ISA is not backwards compatible with the MIPS ISA due to their different instruction sizes and format.

The original format in MIPS of the different instruction types is as follows:

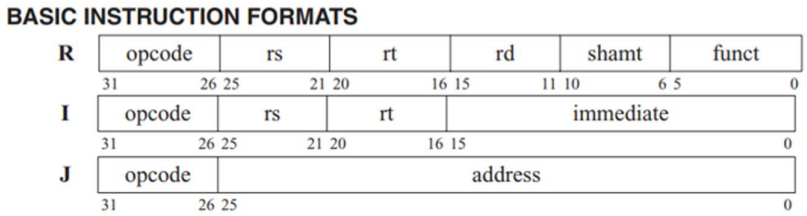

Figure 2: Basic Instruction Format

The table below shows the optimized opcodes for all instructions we used. We based this on the MIPS Instruction Set provided on https://www.dsi.unive.it/~gasparetto/materials/MIPS_Instruction_Set.pdf.

Note that the comments are their comments, since we did not want to lose this information. We added footnotes to explain some things a bit further and keep the column comments as close to the "original" we based our table on. Our disassembled code contains a nop, which has no MIPS opcode, and therefore is not included in table 1 below.

OPCODE	Function	Type	Name	Format Interpertation Example	Comment

000	add	R	add	add R[rd],R[rs],R[rt]
				R[rd]=R[rs]+R[rt]
				add $t0, $t1, $t2
					(1)

001	mul	R	multiplication (without overflow)	mul R[rd],R[rs],R[rt]
				R[rd]=R[rs]*R[rt]
				mul $t0, $t1, $t2
					Result is only 32 bits

010	li	I	load immediate	Li, R[rd], immediate R[rd] = immediate li $t0,100	Pseudo-instruction (provided by assembler1, not processor!) Loads immediate value into register
011	lb	I	load byte	Lw, R[rt],SignExtImm(R[rs]) R[rt] = M[R[rs]+SignExtImm] lb $t0, 100($t1)	transfers one byte of data from main memory to a register.
100	beq	I	branch if equal	Beq, R[rs], R[rt], BranchAddr if(R[rs]==R[rt]) PC=PC+4+BranchAddr beq $t0, $t1,100	Test if registers are equal (4)
101	j	J	jump	J, JumpAddr PC=JumpAddr j 1000	Jump to target address2 (5)

Table 1: Used mips functions, with different opcode

Original notes/comments are shown in brackets, while some of our added notes are not in brackets. The explanation of them can be found on the next page.

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

1 Our assembler: https://cpulator.01xz.net/?sys=mipsr5-spim did not disassemble a li into different functions. Hence we consider this as one function, and keep the implementation as close at the one this assembler uses.

2 We will implement this using a relative jump, with use of beq.

For the average hamming distance we used div, mfhi, mflo we don't take these into account from here on, since they were not relevant for the (total) Hamming Distance calculation.

Assembler Directives:

Directive	Result
.word w	Stores a 24-bit value in a memory word
.byte b	Stores an 8-bit value in a memory word (we only use it indirectly with load byte
.space n	Leave an empty n-byte region of memory for later use
.align n	Align the next datum on a 2^n byte boundary. For example, .align 2 aligns the next value on a word boundary

Table 2: Relevant assembler directives

Registers

To calculate the hamming distance, we suggest using the following registers.:

Register number	Binary encoding	Register name	Description
0	0000	**$zero**	The value 0
1	0001	**$one**	The value 1
2	0010	**$at**	Assembler Temporary
2-9	0011- 1001	**$t1-$t7**	Temporary variables
10-15	1010-1111	**$s1-$s6**	Saved values representing final computed answers/counters

Table 3: Relevant Registers to use

In addition to these changes, we will implement some additional functions, to improve the efficiency.

1. Load immediate was commonly used before an addition, so that in the MIPS code we did not have addi, counting as an extra instruction. Loading an immediate in a register before adding it, could be avoided if we use addi. Hence, we will include this in our reduced ISA.
2. Load immediate was commonly used to load –1, and then add it, instead of using a subtract function. Since we included a register $one, for the storage of the value 1, we only need a sub, and not a subi function.
3. To reduce the number of instruction types, we will implement a jump function using branch if equal. As explained previously in the previous enumeration, all addresses could be represented using 12 bits, as needed for the immediate argument.
4. We worked without delayed branched, meaning that the process would 'wait' until it was decided whether or not to take a branch. The disassembler we used included a nop here, which we want to include as well.

Instruction Format

Since we can reduce the instruction size, to 24 bits, we will use the following encoding of the I and R type instruction.

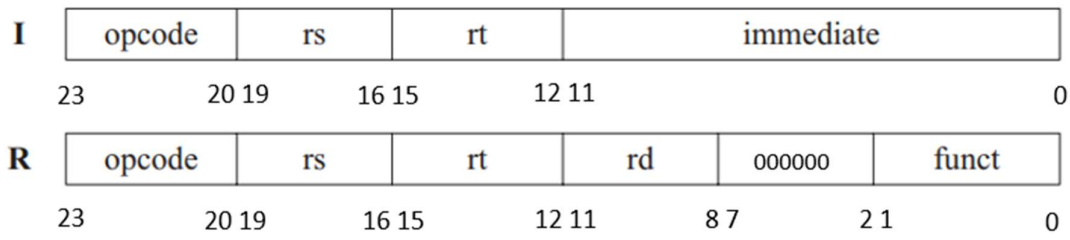

Figure 2: BMIPS instruction format

We could stick with the MIPS standard, of giving all R type instructions an opcode of 00, and specifying the type of R instruction using the funct field, leaving 2 extra bits for the immediate in the I type instructions, or reducing the instruction size to 22 bits. The latter would still need 3 bytes to store the instructions in, and result in 'worse' aligned instructions. Leaving 2 bits extra for the immediate would be possible, but having 12/24 bits for the immediate is consistent with 16/32 bits in mips. Therefore we chose to give each instruction an opcode of 4 bits, where 0000 indicates an R type instruction, so that we can determine the format/encoding simply by checking if the opcode is 0000. We will specify which R type instruction it is in the function field, leaving the shamt field all zeroes, since we don't use functions that need a shamt argument. We will zeropad the opcode for I type instructions with zeroes. Table 4 below shows the new Instruction Set.

OPCODE / funct	Function	Type	Name	Format Interpertation Example	Comment

0000/00	add	R	add	add R[rd],R[rs],R[rt]
				R[rd]=R[rs]+R[rt]
				add $t0, $t1, $t2
					(1)

0000/01	sub	R	subtract	sub R[rd],R[rs],R[rt]
				R[rd]=R[rs]-R[rt]
				sub $t0, $t1, $t2
					(1)

0000/10	mul	R	multiplication (without overflow)	mul R[rd],R[rs],R[rt]
				R[rd]=R[rs]*R[rt]
				mul $t0, $t1, $t2
					Result is only 32 bits

0001	addi	I	add immediate	addi, R[rt],R[rs] immediate R[rt] = R[rs]+ immediate addi $t0, $t1, 100	To load an immediate in $t0, we will use addi $t0, $zero, immediate
0010	lb	I	load byte	Lw, R[rt],SignExtImm(R[rs]) R[rt] = M[R[rs]+SignExtImm] lb $t0, 100($t1)	transfers one byte of data from main memory to a register.
0011	beq	I	branch if equal	Beq, R[rs], R[rt], BranchAddr if(R[rs]==R[rt]) PC=PC+4+BranchAddr beq $t0, $t1,100	Test if registers are equal (4)

Table 4: BMIPS instruction set

Beside these functions, we defined pseudo instructions the user could use. These can be found in table 5 below:

Function	Name	Format Interpertation Example	Assembler translates this to	Comment
li	load immediate	li R[rt], immediate R[rt]=immediate li $t0,12	addi R[rt], $zero, immediate	loads immediate value into a register
j	jump	J, JumpAddr PC=JumpAddr j 1000	beq $zero, $zero, JumpAddr	Jump to target address2 (5)
nop	no operation	nop no operation affecting your process nop	add $zero, $zero, $zero	This is used after beq, to wait for further instructions until a branch is taken or

Table 5: BMIPS pseudo instructions and implementation

In these tables the numbers in brackets are comments from the MIPS cheatsheet provided by the lecturer:

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

Task 4: Datapath

Starting from the diagrams we had during lecture 11:

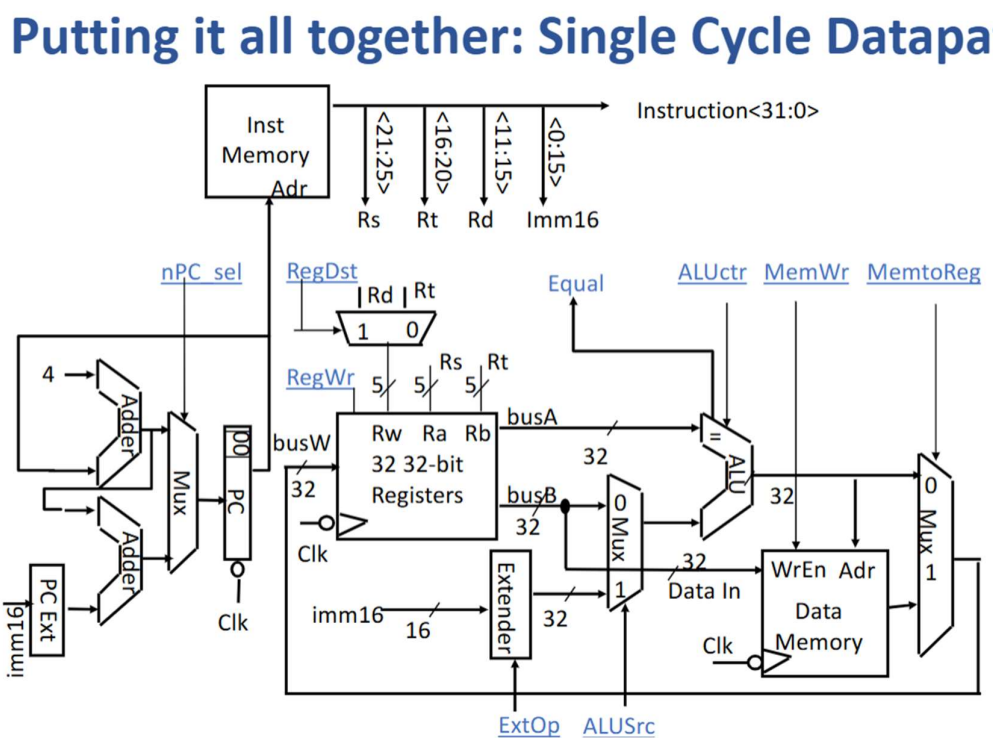

Figure 3: Single cycle datapath from lecture 11

The datapath in figure 3 works for Add, Sub, Addi, Subi, Lw, Sw, and Beq. However, we do not use Subi, Lw and Sw (we do still want to use logical opartions with immediates). To remove Subi as a possibility, we could remove ALUctr, however, this control signal would also be necessary to implement Mul – which we do need. Additionally, we do need Ori, so this ALUctr does need to be (at least) 2 bits to distinguish at least 4 options: Add, Addi, Sub, and Mul. We can remove the connection between busB (coming out of the registers) to Data_In into the data memory, because this connection is only needed for Sw. Technically, we also don't use Lw, but removing this won't make any difference and we might want to use this option if we had an easier way to read user input and store it in memory, but then we would also change our implementation of doing the actual Hamming Distance counting.

Now, the remaining instructions our datapath needs to be able to do, are Li, J, Mul, Lb. But first, let's take a very quick look at our instruction format: which has 24 bits per instruction. The opcode for an instruction is 4 bits and we do not use a lot of bits for R-types: as we do not shift and do not need a separate field for function anymore. Rs = <19:16>, Rt = <15:12>, Rd = <11:8>, Imm12 = <11:0>, func = <1:0>, note that our immediates are 4 bits shorter and that we could give our R-types separate opcodes, as discussed in Task 3, but for quick interpretation of the

instruction we did decide to use a function field also. The function field indicates Add, Sub, or Mul. For the immediates, this means that the “Extender” will now extend from 12 bits to 24 instead of 16 bits to 32. How this impacts the instruction fetch, will be discussed after the remaining instructions that still need to be implemented.

Let’s start with the easiest: Mul. Because Mul only cares about the lower 24 bits of the multiplication and stores them to a register, the ALUctr (to indicate addition vs multiplication) is enough addition to this data path, because our ALU has a 24-bit output. It also has the same structure as Add: Mul rd, rs, rt, so there are no changes necessary at the register block.

Next, we will look at Li, which is a pseudo instruction that our assembler supports. This is often implemented using a Lui instruction followed by an Ori instruction. We will implement this pseudo instruction as Addi $rt, $zero, imm12. Because we do not use any “high” numbers: in fact, for all the numbers we use Li for, do not need more than 12 bits, so we can just zero extend them (because we only use positive numbers). Read appendix A for more information on this topic. We do still need the sign extension option, for branches, but this option is only necessary in the instruction fetch, there “Ext PC” is sign extension, and we can drop the “ExtOp” control signal in the “calculation” path.

Lb, the load byte operation, does sign extension on the byte that is selected from memory. E.g., if at an address the number 0x88776655 is stored, and we have the location of this address in register $t0, then the instruction lb $t1, 0($t0) loads the number 0xffffff88 into register $t1. Recall that we have byte addressable memory. Looking at how the 24-bit instruction is split on the position of bytes in the schematic from the lecture, we can select the first byte in the same manner from the output from the data memory. Then join it again with the original Data_Out with a mux which receives a control signal Load_Size to differentiate Lw and Lb. We do not need to sign extend the Lb here, because we do not work with negative numbers: just with bitstrings, so it might as well be zero extended.

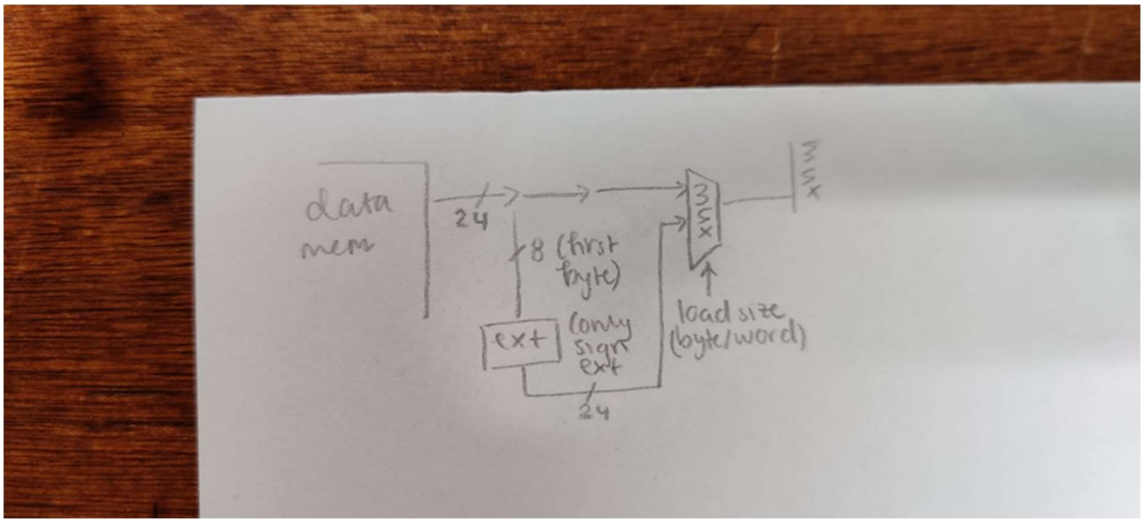

Figure 4: Load byte and load word implementation for BMIPS

Lastly, let’s look at j. We have decided that this is a pseudo instruction, for our ISA. We initially decided to make it a relative jump, where the “address” would be a signed offset of our current address, but as we thought about the size of the jumps that we were making, we realised that that would very well fit within a 12-bit signed immediate. As such, the MIPS instruction J jumpOff(20-bits) will be implemented as Beq

$zero, $zero, jumpOff(12-bits). As such, the program counter incrementation will actually look like the drawing in figure 5 below, where the mux will select regular incrementation or a branch based on the control signal nPC_sel.

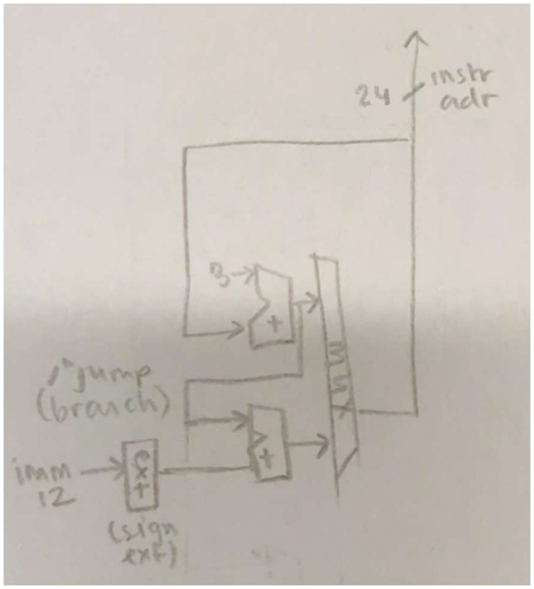

Figure 5: Program counter implementation

Now that we have our own ISA, our instructions are only 3 bytes instead of 4. This changes some things for the instruction fetch. We originally wanted to keep our word size at 32 bits (though if we had more time, we would consider the consequences of changing the word size to 24 bits). We had a bit of leftover time and tried to write down this change as good as possible. We will assume that when we fetch an instruction from memory, it will be 24 bits. The most obvious change is that instead of inputting 4 into the adder in the schematic we used from the lecture shown in Figure 3, we will need to input 3, which we did in the drawing in Figure 5 above. Besides that, nothing else seems to need changing. As such, we will leave these small modifications up to the reader's imagination.

Cache

Caches are divided into blocks, and each block can contain a certain number of bytes. To take advantage of the principle of spatial locality, we will fit more than one byte per block. Our implementation of the HD calculator stores 1 ASCII character per byte, and the minimum length of each bit string (as specified in the assignment) is 5 bits. Thus, to ensure that it's possible to store 1 string in one address, we would need to be able to fit 5 bytes per block. Due to the structure of our ISA, it would be most efficient for our block size to be 24 or 48 bits. Thus, each block will have space to store 6 bits. The number of blocks per cache is also an important factor to consider. The minimum number of sequences (as specified in the assignment) is 4. The number of blocks in a cache is usually a power of 2, and we believe that 4 would not be the most efficient option. Therefore, we will use 8 blocks per cache. Our cache will use a two set-associative mapping because the data we load from the cache will be used to compare two bit strings to each other. This means that two-set associative mapping will be the most efficient for

our system. A further improvement on the ISA is a prefetch instruction, similar to what was mentioned in Task 3 7.a.

Task 5: Low-level implementation

We've interpreted this assignment as having to design a combinational circuit (consisting of elements like gates, finite state machines, and registers) that calculates the hamming distance for two strings of length L (the input) and gives the hamming distance in binary representation of "ceiling function" of ($\log_2(L)$) bits (the output). That basically means that our combinational circuit should XOR the input strings, and then "count the ones" (which might be done with a counter with bit shifting and full adders).

We instead considered implementing the "counting the ones" part, with a Look-Up Table, see the drawing in figure 6 below.

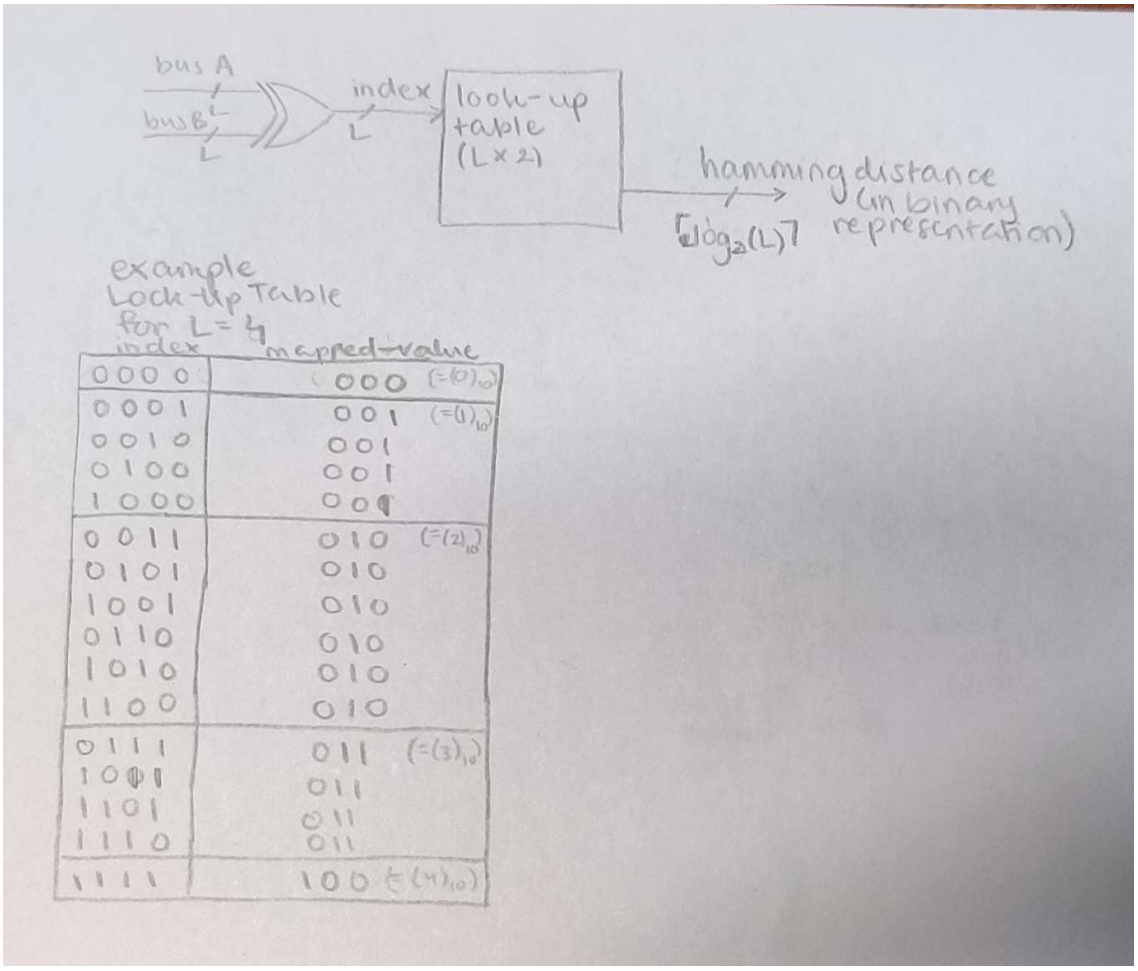

Figure 6: "counting the ones" implementation LUT

If this, however, happens to be not a valid circuit, because we would need to know L before we can hardcode such a table, we could instead consider chopping up the two strings into N many strings of a length, say 8-bits. Then we could hardcode this table for strings of length 8-bits, and

then sequentially, feed the N “counters” through a series of Full Adders (multibit adder), see figure 7 below for an example of 2 strings A and B of length 4 ($C_0 = 0$). Obviously, for more than 2 strings, you would need (“ceiling function” of ($\log_2(L)$)) * 8 many adders.

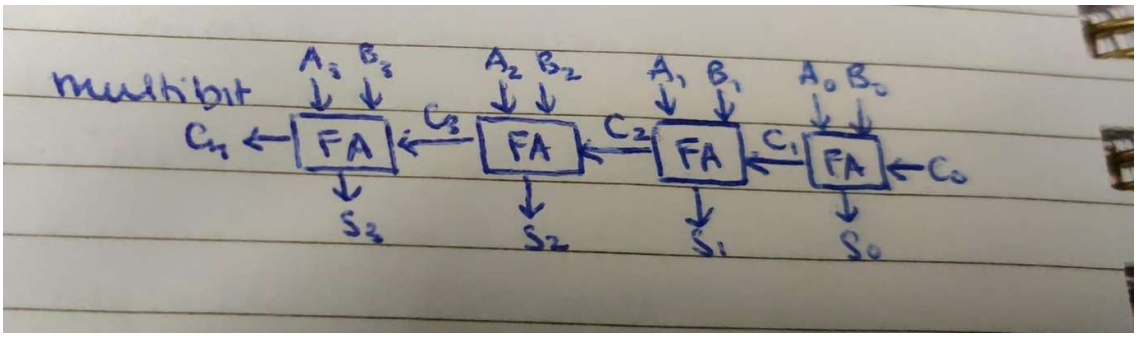

Figure 7: “counting the ones” implementation FA

However, then we will not know how many of such N strings we have to add, and we also do not know how many Full Adders we need to get to an output of “ceiling function” of ($\log_2(L)$) bits. So that does not improve the situation at all.

Lastly, we could consider using clocks and registers: we could store all of the counters into registers. Then we start by adding (using multibit adder) the first two of the N counters (i.e., the mapped value from the look-up table with the index of the XOR result) and writing that to another register that will store the sum. In each clock cycle, we could add the sum with the next counter and store that in the sum. Yet, it does not solve the issue of not knowing how many Full Adders you will need in the circuit, or how many registers are necessary.

In short, those other ideas just seem to overcomplicate things instead of solving the issue our “simple” implementation with the LUT had.

References

MIPS InstructionSet:

https://www.dsi.unive.it/~gasparetto/materials/MIPS_Instruction_Set.pdf

Source for lb:

https://courses.cs.washington.edu/courses/cse378/10sp/lectures/lec02.pdf

Help for nop:
https://www.cs.umd.edu/users/meesh/411/mips-pipe/proj-fall11/mips-doc/node11.html

Mips assembler:

https://cpulator.01xz.net/?sys=mipsr5-spim

Appendix

A: Instruction size determination

We have 43 lines of code for calculating the total Hamming Distance, including the lines only including a label name,
regardless of the number of bitstrings, or the length of a bitstring. In MIPS, this would be a maximum of 43x32 bits program size.

We could define a jump, so that it jumps to an absolute address, not a relative one, which would be indicated with a 26 bits unsigned offset in regular MIPS. This can reach more than the small range we actually use. To design a processor for this part of calculating the Hamming Distance only, 12 bits would suffice, since
43x32 < 64x32 = 2^6x2^5 = 2^11=2^(12-1). So, an address space for a relative jump in our representation could be done with only 12 bits.

The things we use as an immediate, and might prevent us from reducing the size of an immediate are:

1. -1 to decrement a value greater than zero in a register
2. 0 to initialize a counter, or to copy a value to another register with the add function
3. 1 to increment a value in a register
4. A byte
5. An address to jump when the equality for branching holds
6. The number of strings the user wants to enter N (to determine how long you have to keep looping)
7. The number of bytes a bitstring would take up in memory (offset in addresses&to loop over bytes)
   OffsetInAddresses= (X/8 rounded up)*8 where X is the number of characters a bitstring would contain of. To be able to indicate where in memory the next bitstring is located
8. The actual address of the first bitstring (after which each offset later a new one begins)

Explanations of possible limitations

1. We could make a function that decrements a value in a register. Doing this allows us to design an instruction that only uses unsigned integers. And to save memory loads, define a register $one, such that we don't have to load 1 in a temporary register all the time.
2. Loading 0 is not an issue if we decrease the number of bits for the immediate
3. When we have a register $one, always having the value 1, we can simply use an add instruction using this register. But maybe for memory access purposes, we might want to use an add immediate function
4. The immediate provided is a positive offset, always just like in 7. Say that we select M bits for an immediate, then we can represent a number up until(incl) (2^M) –1 , so the user could enter bitstrings of at most length (2^M) –1.
5. The immediate is an address to jump to if branching holds. Say that we have a relative jump, then we could do this with 12 bits only.
6. The number of bitstrings the user wants to enter. If we select S bits for an immediate, the user cannot compare more than (2^S) –1 bitstrings, if memory would allow this many.
7. Same as 4
8. The actual address of the first bitstring. In our MIPS code and assembler, this was 0x00000380, since the first part of memory cannot be used, and we defined some strings before the calculation of the hamming distance. Also, for the place where the first string starts, we need 10 bits to represent this address.

Later we multiply the offset by a number < Nstrings and add it to the address of the first bitstring. But we do this when the values are already loaded in a register. So, we need to be able to load these into a register and do the multiplication/ addition correctly. Which we can calculate.

Based on these restrictions, we believe that reducing the size of the immediate with a few bits shouldn't be a problem. The highest lower bound on the number of bits to assign to the immediate was 12, so we will use that.

Doing this would result in 3bits(opcode) + 2x4bits (register) + 12bits(immediate) = 23 bits in total.

Add that extra bit to the opcode, so that we can implement some other functions/instructions as well, reducing the number of functions to call/ register accesses. -> 24 bits = 3bytes instruction size.

With an immediate size of 12 bits, a user can enter at most to compare 2^11 -1 bitstrings, of 2^11 -1 characters each. The memory needed for this would be ((2^11)-1)^2 = 2^22 + -2^12 +1 = 2^22 – 2^12 +1 bytes. Given that in our assembler, we could start storing bitstrings at 0x00000250, in binary: 1001010000, which is at address 2^9+2^6 + 2^4, which is less than 2^12 -1, we could store all bitstrings if we would have a memory space of 2^22 addresses, meaning 2^22 bytes

the maximum value we would use in a calculation to save in a register, is the address of where the last bitstring starts. Using 24 bit registers would still allow us to store the latest address, since that would be 2^22 – 2^11 +1, which can be represented using 22 bits.