Canvas Group Name:
Student 1: **Student 2:** **Student 3:**

CMPE110 HA2: Human Compiler

Due Date: Friday 02/01/19

Given below is a C code snippet that computes a reduction of a vector consisting of byte-sized values residing in main memory. Note the `uint8_t vs uint32_t`!

RISC-V Manual: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

```
uint8_t v1[256];
uint8_t v2[256];
uint32_t result = 0;
for (int i = 0; i < k; i++) {
      result += v1[i] + v2[i];
}
```

1.) Transform the code snippet above into RISC-V assembly, by implementing a loop that performs k-iterations. Document every line in your assembly code and the variables that each register hold. If you need more than one line of documentation for one instruction, leave the next instruction table cell empty. **(4 Points)**

k is store in $x1
Base address of v1 array is $x2
Base address of v2 array is $x3
Result is temporary store in $x4

Label	Instruction	Documentation
	bge $x0,$x1, Exit	**# Exit if $x1(k)<=0.**
	add $x4,$x0,$x0	**`uint32_t result = 0;` # Initialize rst to 0.**
	add $x5,$x0,$x0	**i = 0; # Initialize i($x5) = 0**
Loop	**add $x6, $x2, $x5**	**# $x6 = base_address_of_array($x2) + i($x5)**
	add $x7, $x3, $x5	**# $x7 = base_address_of_array($x3) + i($x5)**
	lb $x8, 0($x6)	**# load a byte from v1[i]**

	lb $x9, 0($x7)	**# load a byte from v2[i]**
	add $x4, $x4, $x8	**result += v1[i];**
	add $x4, $x4, $x9	**result += v2[i];**
	addi $x5, $x5, 1	**i += 1; # increase i by one**
	bne $x5, $x1, Loop	**# Branch when $x5(i) and $x1(k) is not equal**
Exit		

2.) Assume a non-pipelined processor with a CPI of 6 for load and store instructions, a CPI of 3 for branches and jumps and a CPI of 1 for all other instructions. Compute its average CPI for the code. (k = 16) **(1 Point)**

Initialize part: 1 Branch instruction + 2 add instructions = 1*3 + 2*1 = 5 cycles

Iteration part: 5 add instructions + 2 load/store instructions + 1 branch instruction

= (5*1 + 2*6 + 1*3) * 16 = 320 cycles

Total number of instructions = 3 + (5+2+1)*16 = 131 instructions

CPI = Total number of cycles / Total number of instructions = (320+5)/131 = 2.48

3.) Based on the assumptions of question #2 and a clock frequency of 4 GHz, what is the execution time of your program? **(1 Point)**

Execution time

= Total number of cycles / Clock frequency

= (320+5) / 4E9 = 81.25E-9= 81.25 ns

4.) Optimize your code from question #1 by reducing the number of the **load** operations. Assume that k is always a **multiple** of 16. Before filling out the table below, explain your proposed optimization in two sentences. **(4 Points)**

Optimize by using load word instead of load byte (4 loads -> 1 loads in this case):

Load word load vec[0],vec[1],vec[2] and vec[3],

lw = lb || lb || lb || lb

[Bonus] Optimize by combining two additions (3 times additions -> 2 times in this case)

vec[0] + vec[3] || vec[2] + vec[4] = tmp[high] || tmp[low]
Result = tmp[high] + tmp[low]

k is store in $x1
Base address of v1 array is $x2
Base address of v2 array is $x3
Result is temporary store in $x4

Label	Instruction	Documentation
	bge $x0,$x1, Exit	# Exit if $x1(k)<=0. Can also be a jump to con
	add $x4,$x0,$x0	`uint32_t result = 0;` # Initialize rst to 0.
	add $x5,$x0,$x0	i = 0; # Initialize i($x5) = 0
	addi $x9, $x0, 0xff	
	slli $x9, $x9, 8	
	addi $x9, $x9, 0xff	# mask of lower half word
	slli $10, $x9, 16	# mask of upper half word
	addi $x11, $x0, 0x1ff	
	slli $11, $11, 8	# mask of the 3rd byte with overflow
Loop	add $x6, $x2, $x5	# $x6 = base_address_of_array($x2) + i($x5)
	add $x7, $x3, $x5	# $x7 = base_address_of_array($x3) + i($x5)
v1add	lw $x8, 0($x6)	# load 4 bytes from v1[i]

	and $6, $8, $9	**# get the lower half word**
	and $8, $8, $10	**# get the higher half word**
	srli $8, $8, 16	
	add $8, $8, $6	**# add two half words up**
	andi $6, $8, 0xff	
	and $8, $8, $11	**# also fetch the overflow bit**
	srli $8, $8, 8	
	add $8, $8, $6	**# add two bytes up**
	add $x4, $x4, $x8	**result += v1[i];**
v2add	**lw $x8, 0($x7)**	**# load 4 bytes from v2[i]**
	and $7, $8, $9	**# get the lower half word**
	and $8, $8, $10	**# get the higher half word**
	srli $8, $8, 16	
	add $8, $8, $7	**# add two half words up**
	andi $7, $8, 0xff	
	andi $8, $8, $11	**# also fetch the overflow bit**
	srli $8, $8, 8	
	add $8, $8, $7	**# add two bytes up**
	add $x4, $x4, $x8	**result += v2[i];**
	addi $x5, $x5, 4	**i += 4; # increase i by 4**
Cond	**bne $x5, $x1, Loop**	**# Branch when $x5(i) and $x1(k) is not equal**
Exit		

5.) Based on the assumptions of question #2 what is the average CPI of the code in question #4 now? **(1 Point)**

Initialize part: 1 Branch instruction + 8 add/shift/logic instructions = 1*3 + 8*1 = 11 cycles

Iteration part: 21 add/shifts/log instructions + 2 load/store instructions + 1 branch instruction

= (21*1 + 2*6 + 1*3) * 4 = 144 cycles

Total number of instructions = 1+ 8 + (21+2+1)*4 = 105 instructions

CPI = Total number of cycles / Total number of instructions = (144+11)/105 = 1.48

(Q: is lower CPI better here? A: No. Or we can simply add more NOP instructions. You should really be careful about the context of CPI. It only makes sense when using a particular compiler for a particular program under a particular input. The answer is usually Yes when CPI < 1)
(Q: what's the speedup here?)

6.) Based on the assumptions of question #2 and a clock frequency of 4GHz, what is the execution time of the code in question #4 now? **(1 Point)**

Execution time

= Total number of cycles / Clock frequency

= (144+11) / 4E9 = 38.75E-9= 38.75 ns

7.) Implement the following C-code snippet into RISC-V assembly. Implement two versions, one that uses conditional branch instructions and one that uses a jump table. Assume the jump table resides in main memory. Assume that dir is of type option_t, and has been initialized at some earlier time. **(8 Points)**

```
typedef enum {
OP1,
OP2,
} option_t;

int var = 0;

switch (dir) {
Case OP1:
      var += 3;
Case OP2:
      var += 2;
DEFAULT:
      var += 11;
}
```

`dir is $x1, var is $x2`

Branch instructions (with break/without break):

Label	Instruction	Documentation
	add $x2, $x0, $x0	int var = 0; # initialize var($x2)
	add $x3, $x0, $x0	# initialize the comparator
OP1	bne $x1, $x3, OP2	# op != 0, goto OP2
	addi $x2, $x2, 3	var += 3;
	j Exit	
OP2	addi $x3, $x3, 1	# increase the comparator
	bne $x1, $x3, Default	# op != 1, goto Default
	addi $x2, $x2, 2	var += 2;
	j Exit	
Default	add $x2, $x2, 11	var += 11;
Exit		

`dir is $x1, var is $x2, jump table locates at $x3 (identified by JPT)`

Jump tables (with break)

Label	Instruction	Documentation
	add $x2, $x0, $x0	int var = 0; # initialize var($x2)
	slt $x4, $x1, x0	# $x4 is 1 if x1 < 0
	bne $x4, $x0, Default	# if op < 0, goto Default
	slti $x4, $x1, 2	# $x4 is 1 if x1 < 2
	beq $x4, $x0, Default	# if op >=2, goto Default
	slli $x4, $x1, 2	# else, $x4 = $x1*4
	add $x4, $x4, $x3	# x4 =&(JumpTable[op])

	lw $x5 0($x4)	# $x5 = JumpTable[op]
	jr $x5	# jump to JumpTable[op] # jalr is leagal if PC ($x0) is saved correctly
Exit		
	...	
JPT	addi $x2, $x2, 3	# OP1
	j Exit	# jump back # could be jr
	addi $x2, $x2, 2	# OP2
	j Exit	# jump back # could be jr
Default	addi $x2, $x2, 11	# default, could be after jr
	j Exit	# default jump back # could be jr

Jump tables (without break)

Label	Instruction	Documentation
	add $x2, $x0, $x0	int var = 0; # initialize var($x2)
	slt $x4, $x1, x0	# $x4 is 1 if x1 < 0
	bne $x4, $x0, Default	# if op < 0, goto Default
	slti $x4, $x1, 2	# $x4 is 1 if x1 < 2
	beq $x4, $x0, Default	# if op >=2, goto Default
	slli $x4, $x1, 2	# else, $x4 = $x1*4
	add $x4, $x4, $x3	# x4 =&(JumpTable[op])
	lw $x5 0($x4)	# $x5 = JumpTable[op]
	jr $x5	# jump to JumpTable[op] # jalr is also legal if PC ($x0) is saved correctly
Exit		
	...	
JPT	addi $x2, $x2, 3	# OP1

	addi $x2, $x2, 2	**# OP2**
Default	**addi $x2, $x2, 11**	**# default, could be after jr, but need jal before**
	j Exit	**# default jump back # could be jr**

8.) Consider a CPI of 5 for loads, 3 for branches/jumps and 1 for all other instructions. Compute the average CPI of the 2 implementations. **(4 Points)**

Using the worst case here.
Other instructions:

	#cycles	**#instructions**	**CPI**
Branch-version	**OP1: 2*3+3*1 = 9** **OP2: 3*3+4*1 = 13** **Default: 2*3+4*1=10**	**OP1: 2+3 = 5** **OP2: 3+4 = 7** **Default: 2+4 = 6**	**13 / 6 = 2.17**
JumpTable (with break)	**1*5+4*3+6*1 = 23**	**1+4+6 = 11**	**23 / 11 = 2.09**
JumpTable (without break)	**OP1:1*5+4*3+8*1 = 25** **OP2: 1*5+4*3+7*1 = 24** **Default: 1*5+4*3+6*1 = 23**	**OP1: 1+4+8 = 13** **OP2: 1+4+7=12** **Default: 1+4+6=11**	**25 / 12 = 2.08**

9.) How many switch options are required so that the jump table implementation provides better performance? **(2 Points)**

Assumptions:

- **All the instructions for variable initialization are unnecessary;**
- **Just consider the worst case;**
- **#Options = #all named options + #default options**
- **Use #cycles as the metrics to compare the performance of the same codes;**

Branch-based:

cycles = #options*3+(#options+1)*1 = 4*#options+1

JumpTable (with break)

cycles = 23

JumpTable (without break)

cycles = 22+1*#options

For normal switch flows:

When 4*#options+1 > 23? It's #options > 22/4 = 5.5.

In a non-pipelined computer, jump table provides better performance than branch implementation when the number of options is larger or equal to 6.

For a switch flow without break:

When 4*#options+1 > 22+1*#options? It's only #options > 21/3 = 7.