HW3 - Arman Behnam - A20522906

My Email: abehnam@hawk.iit.edu

2.12

2.12.1

The instruction *add $t0, $s0, $s1* performs an addition of the values stored in registers $s0 and $s1 and then stores the result in register $t0. Given: s0=0x80000000, and s1=0xD0000000.

The result of the addition is: t0=s0+s1. For value of t0.

```
# Given values

s0 = 0x80000000

s1 = 0xD0000000

# Compute the addition

t0 = s0 + s1

t0_hex = hex(t0) t0_hex

RESULT : '0x150000000'
```

The value of t0 after executing the add instruction is 0x150000000. However, if we consider the usual 32-bit MIPS architecture, the result would be truncated to 32 bits. In this case, the lower 32 bits of the result would be kept.

```
# Extracting the lower 32 bits

t0_32bit = t0 & 0xFFFFFFFF

t0_32bit_hex = hex(t0_32bit)

t0_32bit_hex

RESULT: '0x50000000'
```

The value of t0 after truncating to 32 bits is 0x50000000.

2.12.2

In the context of 32-bit arithmetic, "overflow" occurs when the result of an addition does not fit within the 32-bit range. This happens when adding two positive numbers results in a negative value, or adding two negative numbers results in a positive value (considering two's complement representation).

Given: s0=0x80000000, and s1=0xD0000000 (This is also a negative number). The original result was 0x150000000, which doesn't fit in 32 bits. After truncating to 32 bits, we got t0=0x50000000,

which is a positive number. Since the sum of two negative numbers resulted in a positive value, there has been an overflow.

2.12.3

The instruction sub $t0, $s0, $s1 subtracts the value stored in register $s1 from the value stored in register $s0 and then stores the result in register $t0. Given: s0=0x80000000, and s1=0xD0000000

The result of the subtraction is: t0=s0−s1. For the value of t0:

```
# Compute the subtraction

t0_sub = s0 - s1

t0_sub_32bit = t0_sub & 0xFFFFFFFF # Extracting the lower 32 bits

t0_sub_32bit_hex = hex(t0_sub_32bit)

t0_sub_32bit_hex

RESULT: '0xb0000000'
```

The value of t0 after executing the sub instruction and truncating to 32 bits is 0xB0000000. Given: s0=0x80000000 (This is a negative number in two's complement), and s1=0xD0000000 (This is also a negative number). The result t0=0xB0000000 is also a negative number. For this particular operation, the subtraction of two negative numbers has resulted in a negative value. Therefore, there has been no overflow.

2.12.4

We're performing the subtraction: s0−s1 Given: s0=0x80000000, s1=0xD0000000. The result we got was: t0=0xB0000000 (Also a negative number in two's complement)

For our specific case: Both s0 and s1 are negative numbers. Their subtraction resulted in t0, which is also a negative number. Thus, the operation did not violate the two overflow conditions mentioned above. Therefore, there has been no overflow, and the result in t0 is the desired result.

2.12.5

add $t0, $s0, $s1 This adds the values in $s0 and $s1 and stores the result in $t0. Given: s0=0x80000000, and s1=0xD0000000 The result of this addition is: t0=s0+s1.

add $t0, $t0, $s0 This adds the value in $t0 (from the previous operation) with the value in $s0 and stores the result back in $t0. The result of this addition is: t0=t0+s0. The values for each step.

```
# First addition

t0_first_add = s0 + s1

t0_first_add_32bit = t0_first_add & 0xFFFFFFFF # Extracting the lower 32 bits after the first

# Second addition

t0_second_add = t0_first_add_32bit + s0
```

t0_second_add_32bit = t0_second_add & 0xFFFFFFFF # Extracting the lower 32 bits after the second addition

t0_second_add_32bit_hex = hex(t0_second_add_32bit) t0_second_add_32bit_hex

RESULT: '0xd0000000'

After executing the two add instructions, the value of t0 is 0xD0000000.

1. For the first addition (add $t0, $s0, $s1): As we determined earlier, there was an overflow.
2. For the second addition (add $t0, $t0, $s0): The result of the first addition was truncated to 0x50000000. Adding s0 (which is 0x80000000) to that value results in 0xD0000000. Given that we are adding a positive number to a negative number and the result is negative, there is no overflow for this operation.

2.12.6

To determine if there's overflow, we'll consider the entire sequence of instructions and their final effect on $t0: t0=s0+s1, and t0=t0+s0.

Given: s0=0x80000000 (Negative), and s1=0xD0000000 (Negative)

After the first addition: The original result was 0x150000000, which was truncated to 0x50000000 (Positive) due to the 32-bit limitation.

After the second addition: The result is 0xD0000000 (Negative). Here's the sequence of our results:

1. Adding two negative numbers (s0 and s1) gave a positive value (0x50000000), indicating overflow.
2. Adding a negative number (s0) to a positive number (the truncated result from the first addition) gave a negative result (0xD0000000), which is expected and does not indicate overflow.

2.19

2.19.1

sllt2, t0, 44 This is a logical left shift operation. It shifts the value in t0 left by 44 bits and stores the result in t2. However, given that the MIPS architecture is typically 32-bit, a shift by 44 bits effectively means a shift by $44 \bmod 32 = 12$ bits.

Given: t0=0xAAAAAAAA After the shift: $t2 = t0 \ll 12$

or t2, t2, t1 This operation computes the bitwise OR of t2 (from the previous operation) and t1 and stores the result back in t2.

Given: t1=0x12345678 The result is: t2=t2ORt1

The value of t2.

Given register values

```
t0 = 0xAAAAAAAA

t1 = 0x12345678

# Logical left shift

t2_sll = t0 << 12 # Since MIPS is 32-bit, it's effectively a shift by 44 mod 32 = 12 bits

t2_sll_32bit = t2_sll & 0xFFFFFFFF # Extracting the lower 32 bits after the shift

# Bitwise OR operation

t2_or = t2_sll_32bit | t1

t2_or_hex = hex(t2_or)

t2_or_hex

RESULT: '0xbabef678'
```

After executing the sequence of instructions, the value of t2 is 0xBABEF678.

2.19.2

sllt2, t0, 4 This is a logical left shift operation. It shifts the value in t0 left by 4 bits and stores the result in t2. Given: t0=0xAAAAAAAA After the shift: t2=t0≪4

andi t2, t2, −1 The "andi" operation computes the bitwise AND of t2 (from the previous operation) and the immediate value, which is -1 in this case. The "andi" operation is used with a 16-bit immediate value, so the -1 would be represented in 16 bits.

Given the two's complement representation, -1 in 16 bits is represented as all 1s. Thus, the operation will effectively be: t2=t2AND0xFFFF

```
# Logical left shift

t2_sll = t0 << 4

t2_sll_32bit = t2_sll & 0xFFFFFFFF # Extracting the lower 32 bits after the shift

# Bitwise AND operation with immediate value -1 (0xFFFF in 16 bits)

t2_andi = t2_sll_32bit & 0xFFFF

t2_andi_hex = hex(t2_andi)

t2_andi_hex

RESULT: '0xaaa0'
```

After executing the sequence of instructions, the value of t2 is 0xAAA0.

2.19.3

srlt2, t0, 3 This is a logical right shift operation. It shifts the value in t0 right by 3 bits and stores the result in t2. Given: t0=0xAAAAAAAA After the shift: t2=t0≫3

andit2, t2, 0xFFEF The "andi" operation computes the bitwise AND of t2 (from the previous operation) with the immediate value 0xFFEF. The operation will be: t2=t2AND0xFFEF

Logical right shift t2_srl = t0 >> 3 t2_srl_32bit = t2_srl & 0xFFFFFFFF # Extracting the lower 32 bits after the shift # Bitwise AND operation with immediate value 0xFFEF t2_andi = t2_srl_32bit & 0xFFEF t2_andi_hex = hex(t2_andi) t2_andi_hex

RESULT: '0x5545'

After executing the sequence of instructions, the value of t2 is 0x5545.

2.20

1. We first shift the desired bits in t0 to the least significant position and then shift them to bits 31 down to 26.

```
sll $t2, $t0, 15                    # Shift bits 16-11 to bits 31-26 position
```

2. Then clear bits 31 down to 26 in t1. We can do this using a bitwise AND operation with a mask that has 0s in the positions 31 down to 26 and 1s everywhere else.

```
andi $t3, $t1, 0x03FFFFFF                # Clear bits 31-26 in $t1
```

3. Combine the results using a bitwise OR operation.

```
or $t1, $t2, $t3 # Combine the results
```

4. The combined sequence of instructions would be:

```
sll $t2, $t0, 15 andi $t3, $t1, 0x03FFFFFF or $t1, $t2, $t3
```

This sequence extracts bits 16 to 11 from t0 and replaces bits 31 to 26 in t1 with the extracted bits without changing the other 26 bits of t1.

2.22

1. Load the value from the base address of C (i.e., C[0]) into a register.
2. Perform the left shift operation by 4 bits on the loaded value.
3. Store the shifted value back into A.

Here's the MIPS assembly code:

```
lw $t0, 0($s1) # Load C[0] into $t0

sll $t0, $t0, 4 # Perform the left shift by 4 bits on $t0

move $t1, $t0 # Store the shifted value in $t1 (A)
```

In this sequence: $t1 corresponds to A, and $s1 is the base address of C. $t0 is a temporary register used for the operation.

2.23

1. *slt $t2, $0, $t0*: This sets register t2 to 1 if 00 (the constant zero value) is less than t0 (which holds the value 0x00101000), and to 0 otherwise. Since 0 is indeed less than 0x00101000, t2 will be set to 1.
2. *bne $t2, $0, ELSE*: This instruction branches to the label "ELSE" if t2 is not equal to 00. Since we've established that t2 is 1, the program will branch to "ELSE".
3. *j DONE*: This is a jump instruction to the label "DONE". However, because of the previous branch, this instruction is skipped.
4. *ELSE: addi $t2, $t2, 2*: This instruction, under the "ELSE" label, adds 2 to the current value of t2 (which is 1), resulting in t2 being 3.
5. *DONE*: This label marks the end of the sequence.

So, after these instructions, the value of t2 would be 3.

2.26

2.26.1

1. *slt $t2, $0, $t1*: Sets $t2 to 1 if $t1 is greater than 0, else it sets $t2 to 0.
2. *beq $t2, $0, DONE*: Exits the loop if $t2 is 0 (meaning $t1 is not greater than 0).
3. *subi $t1, $t1, 1*: Decreases the value of $t1 by 1.
4. *addi $s2, $s2, 2*: Increases the value of $s2 by 2.
5. j LOOP: Jumps back to the start of the loop.

Given that $t1 is initialized to 10, the loop will iterate 10 times, decrementing $t1 until it reaches 0 and incrementing $s2 by 2 each time. Therefore, the value of $s2 at the end of the loop can be calculated as follows. Each loop iteration adds 2 to $s2, and there are 10 iterations in total. Let's calculate the final value of $s2.

Given that each iteration adds 2 to s2, and knowing there are 10 iterations, we multiply the number of iterations by the value added per iteration:2×10=202×10=20. So, the value of s2 at the end of the loop is 20.

2.26.2

```
int A; // corresponds to $s1, not used in this loop

int B = 0; // corresponds to $s2, initialized to 0 as per your loop

int i = 10; // corresponds to $t1, initialized to 10 as per your loop

int temp; // corresponds to $t2, used for temporary comparisons

// Start of the loop equivalent to the MIPS assembly loop
```

```
while (i > 0) { // This checks the condition set by "slt $t2, $0, $t1" and "beq $t2, $0, DONE"
    i--; // Equivalent to "subi $t1, $t1, 1"
    B += 2; // Equivalent to "addi $s2, $s2, 2"
} // The loop continues until "i" is not greater than 0, which is checked at the start of each iteration
```

This C loop mimics the logic of the assembly code. The loop will continue until i is no longer greater than 0, decrementing i and incrementing B in each iteration, similar to the assembly instructions.

2.26.3

Based on the structure of the loop, five instructions are executed each iteration.

1. The loop starts with t1=N.
2. The loop continues until t1 decrements to 0, which will take N iterations.
3. At the end of the Nth iteration, the "slt" and "beq" instructions are executed, bringing the count to 5N+2 instructions.
4. After the loop, execution jumps to the "DONE" label without executing more instructions.

So, for N iterations, the total number of instructions executed is 5N+2. If you want an exact number, you would need to provide the specific value of N.

2.27

Translating the nested for-loops from C to MIPS assembly:

1. Initialize i to 0.
2. Check if i is less than a; if not, exit the loop.
3. Initialize j to 0.
4. Check if j is less than b; if not, go back to step 1.
5. Calculate the address for D[4*j].
6. Calculate i + j and store the result at the calculated address.
7. Increment j and go to step 4.
8. Increment i and go to step 2.

```
    add $t0, $zero, $zero       # i = 0
outer_loop:
    slt $t2, $t0, $s0 # if i >= a, exit the loop
    beq $t2, $zero, exit_outer
    add $t1, $zero, $zero # j = 0
inner_loop:
    slt $t2, $t1, $s1           # if j >= b, exit the loop
```

```
        beq $t2, $zero, update_i
        sll $t3, $t1, 2          # t3 = 4 * j (calculate the offset for D[4*j])
        add $t3, $s2, $t3        # t3 = address of D[4*j]
        add $t4, $t0, $t1        # t4 = i + j
        sw $t4, 0($t3)           # D[4*j] = i + j
        addi $t1, $t1, 1         # j++
        j inner_loop
update_i:
        addi $t0, $t0, 1 # i++
        j outer_loop
exit_outer:
```

This code assumes that the array D is an array of integers (since we're using an offset of 4*j, typical of 4-byte integers) and that the base address of D is in $s2. The code uses $t0 and $t1 for i and j, respectively, and $s0 and $s1 for a and b, respectively. It also uses additional temporary registers $t2, $t3, and $t4 for comparisons and calculations.

2.28

The instructions:

- Initialization and comparison instructions: 2 (add for initialization, slt for comparison)
- Branching to exit the loop: 1 (beq)
- Instructions inside the inner loop (including address calculation, addition, memory store, increment, and unconditional jump): 7
- Instructions for incrementing i and unconditional jump to continue the outer loop: 2

This totals 12 unique instructions in the loop's implementation.

Given a = 10 and b = 1, the inner loop executes once for each iteration of the outer loop, which iterates 10 times. Here's the breakdown:

- The outer loop runs 11 times (10 iterations plus the one failing the condition).
- Each iteration of the outer loop involves 1 comparison and 1 branch instruction, and it initializes j (1 instruction), total 3 instructions per iteration, resulting in 3×11=333×11=33 instructions.
- The inner loop runs 10 times (once for each outer iteration) and involves 7 instructions (as listed above), total 7×10=707×10=70 instructions.
- The last iteration that checks j < b and increments i contributes 2 instructions per outer loop iteration, resulting in 2×11=222×11=22 instructions.

Adding these up gives the total number of executed instructions.

```
# Number of iterations for each loop

outer_loop_iterations = 11 # 10 successful iterations + 1 final failing check

inner_loop_iterations = 10 # Because 'b' is 1, so it runs once for each outer iteration

# Number of instructions executed per iteration

instructions_outer = 3 # (slt, beq, add for 'j')

instructions_inner = 7 # All instructions inside the inner loop

instructions_post_outer = 2 # (addi for 'i', j to continue the loop)

# Calculating the total number of instructions executed

total_instructions = (outer_loop_iterations * instructions_outer) + \ (inner_loop_iterations * instructions_inner) + \ (outer_loop_iterations * instructions_post_outer)

total_instructions

RESULT: 125
```

The total number of MIPS instructions executed to complete the nested loops, given that a = 10 and b = 1, is 125.

2.38

Given the 32-bit hexadecimal value 0x11223344 at address 0x10000000, the byte layout in memory would be:

- 0x10000000: 0x11 (MSB)
- 0x10000001: 0x22
- 0x10000002: 0x33
- 0x10000003: 0x44 (LSB)

The given MIPS instructions perform the following operations:

lbu $t0, 0($t1): This instruction loads the byte at the address contained in register $t1 (0x10000000) into register $t0. Because it's a "load byte unsigned" (lbu) instruction, it will retrieve the byte at that address without sign extension. Given the big-endian memory organization, the byte loaded into $t0 is 0x11.

sw $t0, 0($t2): This instruction stores the word in register $t0 at the address contained in register $t2 (0x10000010). Since $t0 was populated by the "lbu" instruction and holds the value 0x11, this value is stored into the word at the new address. However, because it's a "store word" (sw) instruction, it deals with words (4 bytes), not single bytes. The 0x11 value in $t0 will be stored in the least significant byte of the word at the address in $t2, while the remaining more significant

bytes will be zeroes, resulting in the word 0x00000011 being stored at 0x10000010. So, the value stored at the address pointed to by register $t2 (0x10000010) is 0x00000011.

2.39

To create the 32-bit constant 0010 0000 0000 0001 0100 1001 0010 0100_2 and store it in register t1, we would first convert this binary number to hexadecimal for easier handling in MIPS assembly, and then use the appropriate instructions to load this value into the register.

Convert the binary number to hexadecimal:

0010 0000 0000 0001 0100 1001 0010 0100_2 equals 0x20014924 in hexadecimal.

In MIPS, there is no single instruction that directly loads a 32-bit immediate value into a register. Instead, we use a combination of instructions to achieve this. One common approach is to use the lui (load upper immediate) and ori (bitwise or immediate) instructions.

```
lui $t1, 0x2001 # Load the upper 16 bits of the constant into $t1
```

```
ori $t1, $t1, 0x4924 # Bitwise-OR to insert the lower 16 bits into $t1
```

The lui instruction shifts the 16-bit immediate value left by 16 bits and stores the result in register t1, effectively loading the upper half of the 32-bit constant. The ori instruction then inserts the lower 16 bits. The combination of these two instructions effectively constructs the full 32-bit constant in register t1.

2.46

2.46.1

To determine if this is a good design choice, we need to calculate the total execution time before and after the proposed changes and then compare these times.

1. Before the changes:
    a. Calculate the total number of cycles for each type of instruction and then sum them to get the total number of cycles for the program.
    b. Use the clock cycle time to find the total execution time.
2. After the changes:
    a. Reduce the number of arithmetic instructions by 25%.
    b. Adjust the CPI for the arithmetic instructions due to the increased complexity (as the clock cycle time increases by 10%, the CPI for arithmetic instructions might increase correspondingly).
    c. Calculate the new total number of cycles and the new execution time using the increased clock cycle time.
3. Comparison:
    a. Compare the execution times before and after the changes to evaluate the effectiveness of the design choice.

```
# Constants for the initial scenario
```

```
initial_arithmetic_instructions = 500e6

initial_load_store_instructions = 300e6

initial_branch_instructions = 100e6

CPI_arithmetic = 1

CPI_load_store = 10

CPI_branch = 3

# Total cycles calculation before the changes

total_cycles_before = (initial_arithmetic_instructions * CPI_arithmetic + initial_load_store_instructions * CPI_load_store + initial_branch_instructions * CPI_branch) # Assumptions for the new scenario

reduction_in_arithmetic_instructions = 0.25 # 25% reduction

increase_in_clock_cycle_time = 0.10 # 10% increase

# New number of arithmetic instructions

new_arithmetic_instructions = initial_arithmetic_instructions * (1 - reduction_in_arithmetic_instructions)

# Total cycles calculation after the changes (assuming CPI for arithmetic operations doesn't change)

total_cycles_after = (new_arithmetic_instructions * CPI_arithmetic + initial_load_store_instructions * CPI_load_store + initial_branch_instructions * CPI_branch)

# The execution time is directly proportional to the number of cycles and the clock cycle time.

# We assume an initial arbitrary clock cycle time to make this comparison, as we're comparing the proportionality.

arbitrary_clock_cycle_time = 1 # This can be any non-zero number since we're comparing the ratio.

execution_time_before = total_cycles_before * arbitrary_clock_cycle_time

execution_time_after = total_cycles_after * (arbitrary_clock_cycle_time * (1 + increase_in_clock_cycle_time))

# We are interested in the change in execution time, not the exact execution time.

change_in_execution_time = execution_time_after - execution_time_before

total_cycles_before, total_cycles_after, change_in_execution_time

RESULT :(3800000000.0, 3675000000.0, 242500000.00000048)
```

Before the changes, the program requires 3,800,000,000 cycles to complete. After implementing the more powerful arithmetic instructions and increasing the clock cycle time by 10%, the program requires 3,675,000,000 cycles.

However, due to the increase in clock cycle time, the actual execution time increases by 242,500,000 units (in the same time unit used for the clock cycle time).

So, while the total number of cycles has decreased, the execution time has increased due to the longer clock cycle. This is crucial because the user experiences execution time, not the number of cycles.

Therefore, this might not be a good design choice if the goal is to decrease execution time. The 25% reduction in arithmetic instructions does not compensate for the 10% increase in clock cycle time. In performance-sensitive applications, this trade-off could lead to a slower perceived performance, even though the processor is performing fewer overall computations.

2.46.2

Here, the enhancement is doubling the performance of arithmetic instructions, meaning they would take half the cycles per instruction (CPI) as before. First, we'll calculate the initial execution time based on the CPI values and the number of instructions. Then, we'll calculate the new execution time after the improvement and determine the speedup from these values.

1. Initial Execution Time (ET_initial): This is calculated as the product of the total cycles (calculated from the CPI and the number of instructions) and the clock cycle time (CCT). We assume CCT remains constant, where Total Cycles is the sum of the products of CPI and the instruction count for each type of instruction.
2. New Execution Time (ET_new): After the enhancement, arithmetic instructions are executed faster (half the CPI or one-tenth the CPI, depending on the scenario). We recalculate the total cycles with the new CPI for arithmetic instructions and then determine the new execution time using the same CCT.
3. Speedup: This is the ratio of the initial execution time to the new execution time.

```
# Constants for the initial scenario

CPI_arithmetic_initial = 1 # Initial CPI for arithmetic instructions

# For the scenario of doubling the performance (half the CPI)

CPI_arithmetic_double = CPI_arithmetic_initial / 2 # CPI is halved

# For the scenario of a tenfold increase in performance (one-tenth the CPI)

CPI_arithmetic_tenfold = CPI_arithmetic_initial / 10

# CPI is one-tenth # Total cycles calculation before the changes (initial scenario)

total_cycles_initial = (initial_arithmetic_instructions * CPI_arithmetic_initial + initial_load_store_instructions * CPI_load_store + initial_branch_instructions * CPI_branch)
```

```
# Total cycles calculation after doubling arithmetic performance

total_cycles_double = (initial_arithmetic_instructions * CPI_arithmetic_double + initial_load_store_instructions * CPI_load_store + initial_branch_instructions * CPI_branch)

# Total cycles calculation after tenfold increase in arithmetic performance

total_cycles_tenfold = (initial_arithmetic_instructions * CPI_arithmetic_tenfold + initial_load_store_instructions * CPI_load_store + initial_branch_instructions * CPI_branch)

# Speedup calculation: It's the ratio of the number of cycles before and after the improvement.

# This works under the assumption that clock cycle time remains constant.

speedup_double = total_cycles_initial / total_cycles_double

speedup_tenfold = total_cycles_initial / total_cycles_tenfold

speedup_double, speedup_tenfold

RESULT :(1.0704225352112675, 1.1343283582089552)
```

Doubling the performance of arithmetic instructions (i.e., halving the CPI for these instructions) results in an overall speedup of approximately 1.07. This means the machine would be about 7% faster with the enhanced arithmetic instructions.

Improving the performance of arithmetic instructions tenfold (i.e., making the CPI one-tenth of its original value) results in an overall speedup of approximately 1.13. This means the machine would be about 13% faster with these highly enhanced arithmetic instructions.

2.47

2.47.1

The average CPI (Cycles Per Instruction) is calculated based on the individual CPIs of different types of instructions and their respective proportions in the total instruction mix.

Given: Arithmetic instructions: 70% of total, 2 cycles each, Load/Store instructions: 10% of total, 6 cycles each, and Branch instructions: 20% of total, 3 cycles each.

```
# Percentages of each type of instruction

percentage_arithmetic = 0.70

percentage_load_store = 0.10

percentage_branch = 0.20

# Cycles per instruction for each type of instruction

cycles_arithmetic = 2

cycles_load_store = 6
```

```
cycles_branch = 3

# Calculating the average CPI

average_CPI = (percentage_arithmetic * cycles_arithmetic + percentage_load_store * cycles_load_store + percentage_branch * cycles_branch)

average_CPI
```

RESULT: 2.6

The average CPI (Cycles Per Instruction) given the instruction mix and individual cycles for each type of instruction is 2.6.

2.47.2

To achieve a 25% improvement in performance, we need to understand that performance is inversely related to execution time, which is directly proportional to both the CPI and the number of instructions. If we want to improve performance by 25%, we're effectively saying we want to reduce the execution time to 75% of its current value.

Given that we are only improving the arithmetic instructions and not the load/store or branch instructions, we need to focus on the impact that the arithmetic instructions have on the overall CPI and, therefore, the execution time. The formula for the new average CPI, given a 25% improvement in performance, can be derived from the relationship:

New Execution Time=0.75×Old Execution TimeNew Execution Time=0.75×Old Execution

Since execution time is directly related to the product of CPI and the instruction count (and the instruction count remains constant), we can say:

New Average CPI=0.75×Old Average CPINew Average CPI=0.75×Old Average CPI

We have already established that the average CPI with the old performance is 2.6. Therefore, the new average CPI would be 0.75×2.60.75×2.6.

Now, we want to find out the new CPI for arithmetic instructions that would give us this new average CPI. The average CPI is composed of the contributions from each type of instruction. The contribution from the arithmetic instructions is what we are allowed to change. The maximum CPI that the improved arithmetic instructions can have to achieve this overall performance improvement.

```
# The target is a 25% improvement in performance, which means the new execution time should be 75% of the old execution time.

target_execution_time_factor = 0.75

target_average_CPI = target_execution_time_factor * average_CPI

fixed_CPI_contribution = (percentage_load_store * cycles_load_store + percentage_branch * cycles_branch)
```

new_CPI_arithmetic = (target_average_CPI - fixed_CPI_contribution) / percentage_arithmetic

new_CPI_arithmetic

RESULT

1.0714285714285714

To achieve a 25% improvement in overall performance, the new cycles per arithmetic instruction must be reduced, on average, to approximately 1.07. This means the arithmetic instructions need to execute nearly twice as fast as they did previously, assuming the performance of load/store and branch instructions remains unchanged.

2.47.3

A 50% improvement in performance means we want to make the program run in half the time it currently takes. This is equivalent to saying that we want the new execution time to be 50% of the current execution time, or conversely, that we want a performance increase of 100% from the current state.

Given the direct relationship between execution time and CPI (since the number of instructions does not change), achieving a 50% reduction in execution time implies we need a corresponding reduction in the average CPI.

The new average CPI target with a 50% improvement in performance, and then find out what the new CPI for arithmetic instructions needs to be to achieve this new average CPI, assuming the CPIs for load/store and branch instructions remain unchanged.

The target is a 50% improvement in performance, which means the new execution time should be 50% of the old execution time.

target_execution_time_factor_50 = 0.50

Calculate the new target average CPI to meet the desired performance improvement.

target_average_CPI_50 = target_execution_time_factor_50 * average_CPI

new_CPI_arithmetic_50 = (target_average_CPI_50 - fixed_CPI_contribution) / percentage_arithmetic

new_CPI_arithmetic_50

RESULT :0.14285714285714268

To achieve a 50% improvement in overall performance, the new cycles per arithmetic instruction must be reduced, on average, to approximately 0.143. This means the arithmetic instructions need to execute significantly faster than they did previously, assuming the performance of load/store and branch instructions remains unchanged.

Furthermore:

2.13

2.13.1

Given s0 holds the value 1281012810: If s0 and s1 are both positive and their sum is negative, then overflow has occurred. If s0 is positive and s1 is negative (or vice versa) and their sum is positive, then overflow has occurred. But since s0 is positive and relatively small, it's impossible for its addition with a negative number to overflow in a 32-bit system.

For the first case, the maximum positive value for a 32-bit integer in two's complement representation is 231−1231−1 (since one bit is reserved for the sign). To determine the range for s1 that causes an overflow, we need to find values for which the sum s0+s1 exceeds this maximum.

The overflow for the addition of two positive numbers will occur if the sum exceeds the maximum positive 32-bit two's complement value. Given: s0=128 max positive=231−1max positive=231−1

```
# Given value for s0

s0_value = 128

# Maximum positive value for a 32-bit integer in two's complement

max_positive_32bit = 2**31 – 1

# Calculate the smallest value of s1 that causes overflow

min_s1_overflow = max_positive_32bit - s0_value + 1

# The range for s1 that would cause an overflow is [min_s1_overflow, 2**31)

range_s1_overflow = (min_s1_overflow, 2**31)

range_s1_overflow
```

RESULT: (2147483520, 2147483648)

The range of values for s1 that would result in overflow when added to s0 (which has a value of 12810128_{10}) is: [2147483520,2147483648)

This means any value for s1 within this range, when added to s0, will cause an overflow in a 32-bit system.

2.13.2

Given s0=128 (which is positive):

1. For the first case, there will be no overflow since s0 is small, and subtracting any negative number from it will not cause the result to overflow in a 32-bit system.
2. For the second case, the condition for overflow is: s0−s1<minimum 32-bit integer Where the minimum 32-bit integer in two's complement representation is −231−231. From this, we can determine the range of values for s1 that would cause an overflow.

```
# Minimum 32-bit value in two's complement
```

```
min_32bit = -2**31

# To determine the maximum value of s1 that causes overflow:

# s0 - s1 < min_32bit

# => s1 > s0 - min_32bit

max_s1_overflow = s0_value - min_32bit

# The range for s1 that would cause an overflow is [max_s1_overflow, 2**31)

# as values equal to or above 2**31 are not valid 32-bit two's complement integers.

range_s1_overflow_subtraction = (max_s1_overflow, 2**31)

range_s1_overflow_subtraction

RESULT: (2147483776, 2147483648)
```

For the subtraction instruction sub t0, s0,s1, the range of values for s1 that would result in overflow when subtracted from s0 (which has a value of 1281012810) is: [2147483776,2147483648)

This means any value for s1 within this range, when subtracted from s0, will cause an overflow in a 32-bit system. However, note that the range is somewhat counter-intuitive since the start of the range is larger than the end. This is because the value 231231 (2147483648) is not a valid 32-bit two's complement integer, so any value equal to or above this would be invalid. Thus, the only valid value in this range that would cause an overflow is s1=2147483776.

2.13.3

Given s0=128 (positive):

1. For the first case: The condition for overflow is s1−s0>max positive 32-bit integer where the max positive 32-bit integer in two's complement representation is 231−1231−1.
2. For the second case: There will be no overflow since s0 is positive and small, subtracting it from any positive number will not cause the result to overflow in a 32-bit system.

```
# Given value for s0

s0_value = 128

# Maximum positive value for a 32-bit integer in two's complement

max_positive_32bit = 2**31 – 1

# Calculate the minimum value of s1 that causes overflow for the reversed subtraction

min_s1_overflow_subtraction_reverse = s0_value + max_positive_32bit + 1

# The range for s1 that would cause an overflow is (-2**31, min_s1_overflow_subtraction_reverse]

range_s1_overflow_subtraction_reverse = (-2**31, min_s1_overflow_subtraction_reverse)
```

range_s1_overflow_subtraction_reverse

RESULT: (-2147483648, 2147483776)

For the instruction subt0, s1, s0, the range of values for s1 that would result in overflow when s0 (which has a value of 1281012810) is subtracted from s1 is: (−2147483648,2147483776]

This means any value for s1 within this range, when s0 is subtracted from it, will cause an overflow in a 32-bit system.