PES University, Bangalore

(Established under Karnataka Act No.16 of 2013)

RISC-V Architecture

(UE21EC352A)

Project Report

FRACTIONAL KNAPSACK ALGORITHM

Team Members:

Srinivasa Puranika Bhatta	(PES1UG21EC295)
Sumukh Shadakshari	(PES1UG21EC302)

SEM: 5

Section: ‘F’

Contents

Problem Statement:

Simple Implementation of the solution for the Fractional Knapsack Problem using the Greedy approach with a RISC-V Assembly Level Program.

Introduction:

The Knapsack Problem is a classic optimization problem in computer science and mathematics that involves selecting items to maximize the total profit within a limited capacity. The problem is named after the idea of a knapsack or backpack where items have different profits and weights, and the goal is to determine the combination of items that maximizes the profit while not exceeding the weight capacity of the knapsack. Fractional Knapsack indicates that a fraction of an item can be used to fill the knapsack if necessary.

Formulation:

Given a set of items, each with a specific weight and profit, and a knapsack or container with a maximum weight capacity, the problem is to determine the most valuable combination of items that can be included in the knapsack without exceeding its capacity.

Types of Knapsack Problems:

1. Fractional Knapsack Problem: In this version, items can only be selected in whole numbers (either an item is selected entirely or not at all). It's a binary decision—either an item is included or excluded.

2. Fractional Knapsack Problem: Here, fractions of items can be taken. Unlike the Fractional Knapsack, items can be divided and included in portions, making it possible to take a fraction of an item to maximize the profit.

The Knapsack Problem has numerous real-world applications across various domains:

1. Resource Allocation: It's used in resource allocation scenarios where there are limited resources (like budget, space, weight, etc.) and multiple options (items) competing for these resources. Examples include maximizing profit with limited inventory space or optimizing a portfolio given a limited investment budget.

2. Computer Algorithms: The Knapsack Problem is a fundamental problem in computer science and is used to analyze and design algorithms, particularly in the field of dynamic programming and combinatorial optimization.

3. Finance and Investment: In finance, it can be applied to portfolio optimization, where an investor aims to maximize the return on a portfolio while considering the limited investment capital available.

4. Operations Research: It finds applications in operations research for optimizing processes, such as load balancing in networks, scheduling problems, and optimizing resources in supply chain management.

5. Genetics and Biology: In genetics and biology, it can be applied to problems like genome sequencing and protein structure prediction.

Fractional Knapsack Problem:

The Fractional Knapsack problem using Greedy approach can be defined as follows:

*The basic idea of the greedy approach is to calculate the ratio **profit/weight** for each item and sort the item based on this ratio. Then take the item with the highest ratio and add them as much as we can (can be the whole element or a fraction of it).*

This will always give the maximum profit because, in each step it adds an element such that this is the maximum possible profit for that much weight.

A Simple Example:

*Consider the example: **Profit = [100,60,120], Weights = [20, 10, 30], Capacity = 50**.*

***Sorting:** Initially sort the array based on the profit/weight ratio. The sorted arrays will be [60,100,120], [10,20,30].*

<u>Iteration:</u>

- *For **i = 0**, weight = 10 which is less than Capacity. So add this element in the knapsack. **profit = 60** and remaining **capacity = 50 – 10 = 40**.*
- *For **i = 1**, weight = 20 which is less than Capacity. So add this element too. **profit = 60 + 100 = 160** and remaining **Capacity = 40 – 20 = 20**.*
- *For **i = 2**, weight = 30 is greater than Capacity. So add 20/30 fraction = **2/3** fraction of the element. Therefore **profit** = 2/3 * 120 + 160 = 80 + 160 = **240** and remaining **Capacity** becomes **0**.*

*So the final profit becomes **240** for **Capacity = 50**.*

Note: Detailed manual calculations are provided before the code.

Algorithm:

The code designed needs the items to be sorted based on the descending order of their profit per unit weight. The given assembly code serves two main functions: first, it performs a bubble sort on the 'ppw' that is the profit per weight array in descending order while simultaneously reordering the 'weights' array and the ‘profits’ correspondingly, and second, it implements the Fractional Knapsack problem-solving algorithm using the sorted 'profits' and 'weights' arrays using the greedy approach.

Bubble Sort Algorithm for 'ppw' Array in Descending Order:

1. **Outer Loop**:
   - Initializes an outer loop counter to the number of elements in the array minus one.
   - Iterates through the array from the beginning to the second-to-last element.
2. **Inner Loop**:
   - Initializes an inner loop counter to zero.

- Compares adjacent elements in the 'ppw' array.
- If the current element is smaller than the adjacent element, it swaps the elements in the 'ppw', ’profits’ and 'weights' arrays.
- The ‘ppw’ array only holds the quotient part of the profit / weight. The remainder is held in another array ‘rem’.
- If the profits in the ‘ppw’ array are equal for two adjacent elements then theirs remainders are used to determine which element is bigger, i.e.., if the current element has a remainder larger than the adjacent element then it is swapped with its adjacent element.

3. **Sorting Process**:
   - Continues this process until the 'ppw' array is sorted in descending order.
   - Once sorted, the arrays are prepared for the Fractional Knapsack problem by having the 'ppw' array sorted while maintaining the correspondence with the 'weights' array and the ‘profit’ array.

Fractional Knapsack Problem-Solving Algorithm:

1. **Initialization**:
   - Sets up pointers to the sorted 'profit' and 'weights' arrays.
   - Initializes loop counters and variables to track the current capacity and the resulting knapsack profit.
2. **Knapsack Algorithm**:
   - Iterates through the sorted arrays.
   - Loads the profit and weight of the current item and calculates the remaining capacity in the bag.
   - Checks if the current item can fit in the bag based on its weight and the remaining capacity.
   - If the remaining capacity is greater than the total weight of the current item, updates the knapsack profit and the current capacity by adding the complete weight to the knapsack.
   - Otherwise, it adds only the fraction of the current weight which can be accommodated in the knapsack and updates the profit and weight accordingly
3. **Printing the Result**:
   - Once the algorithm completes, it prints the string "ANSWER\n" and then prints the calculated knapsack profit using ‘ecall’.

Short note about ‘ecall’:

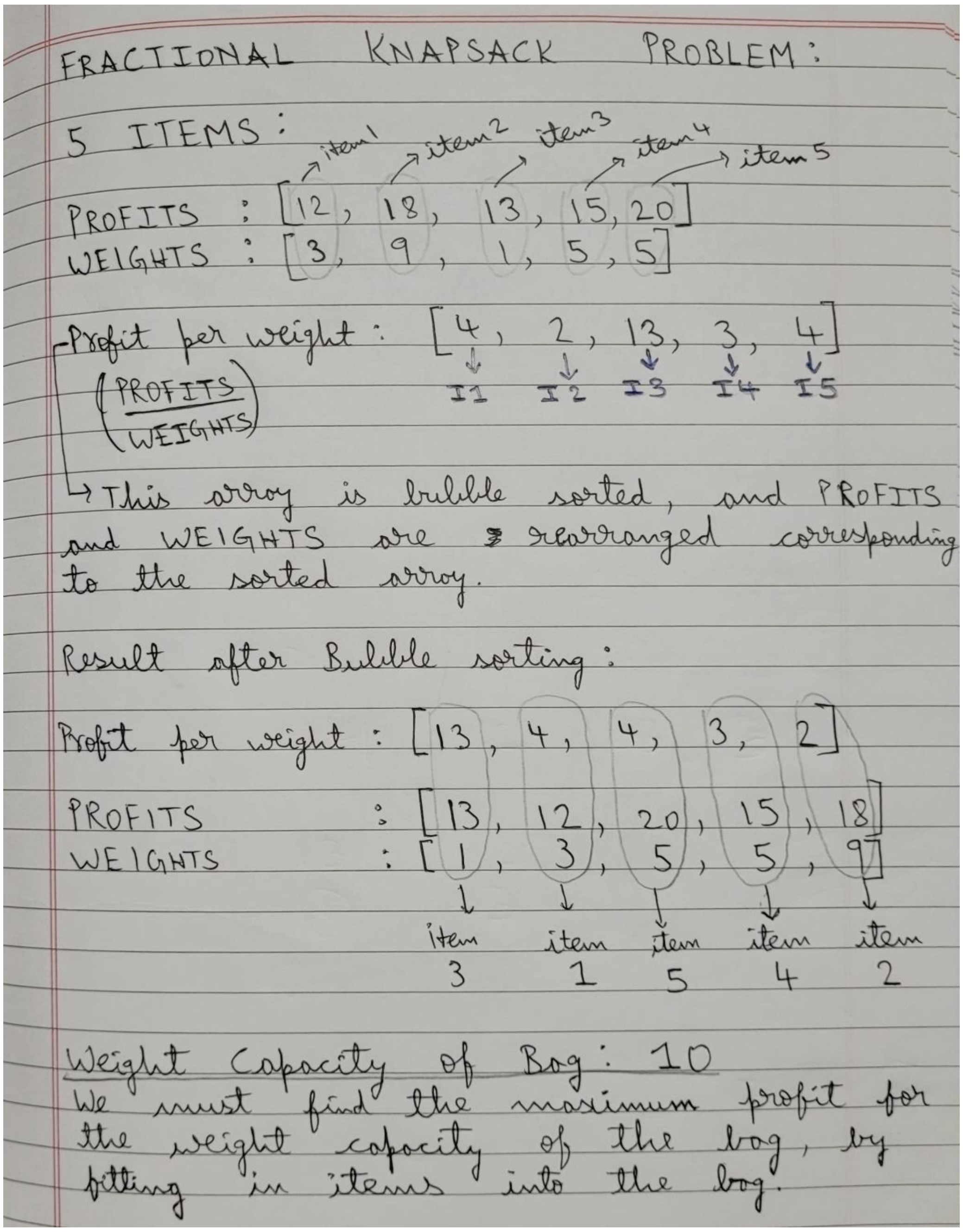
‘ecall’ is an instruction used for making a system call. It's a privileged instruction that allows a RISC-V program running in user mode to request services from the operating system or the underlying execution environment.

System calls are essential for user programs to interact with the operating system kernel and request various privileged operations, such as I/O operations (like printing to the console), memory allocation, file operations, and more.

The ‘ecall’ instruction generates a trap, switching the processor's execution mode from user mode to a higher privilege level (such as supervisor mode) to perform the requested system call. The specific system call to execute is determined by the values in specific registers, typically a7 (argument register) that contains the system call number and other registers holding arguments or results for the specific system call.

RISC-V Assembly Language Program for Fractional Knapsack Algorithm:

Manual Calculation of the Problem:



FRACTIONAL KNAPSACK PROBLEM:

5 ITEMS: item1 item2 item3 item4 item5

PROFITS : [12, 18, 13, 15, 20]
WEIGHTS : [3, 9, 1, 5, 5]

Profit per weight : [4, 2, 13, 3, 4]
(PROFITS/WEIGHTS) I1 I2 I3 I4 I5

→ This array is bubble sorted, and PROFITS and WEIGHTS are rearranged corresponding to the sorted array.

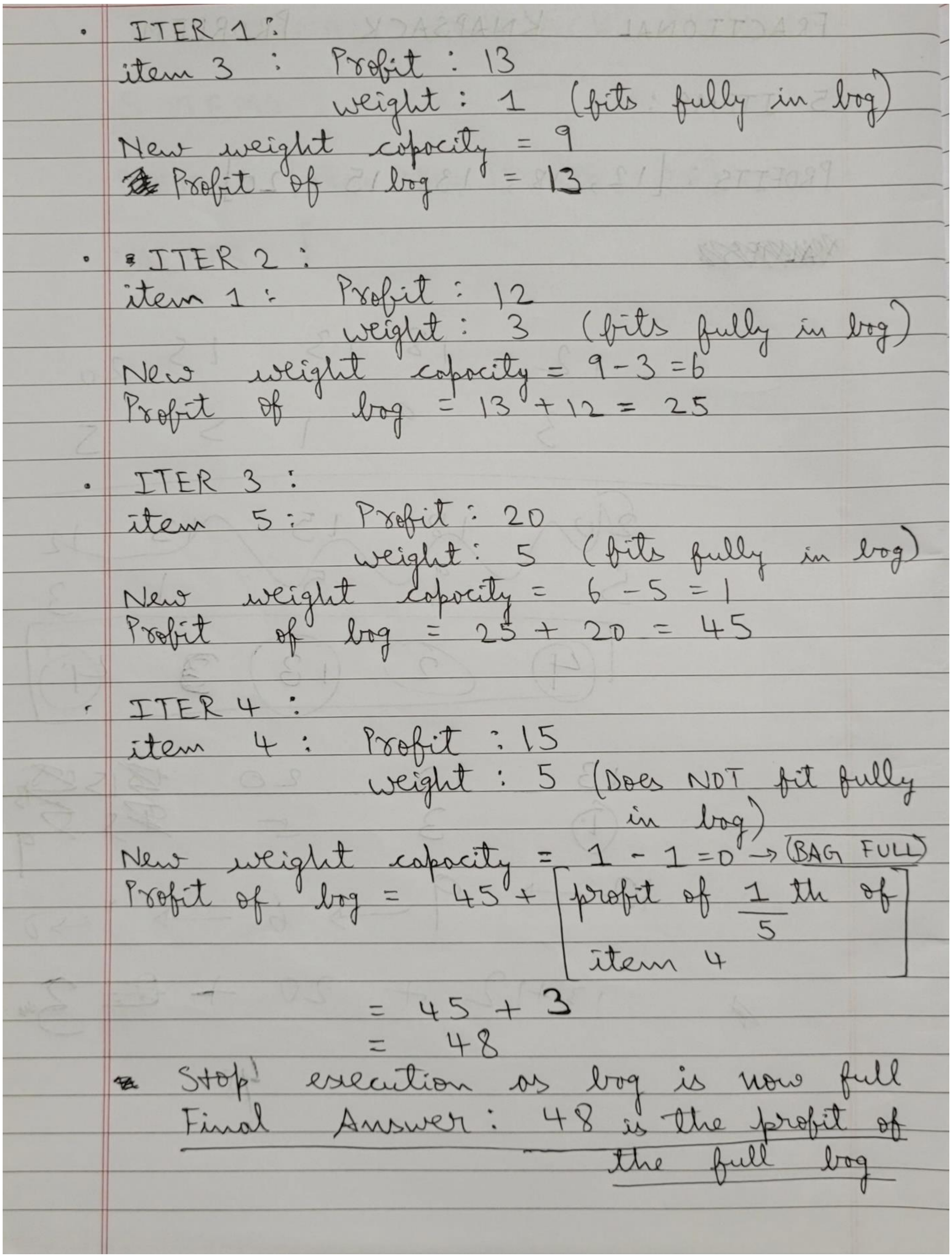
Result after Bubble sorting:

Profit per weight : [13, 4, 4, 3, 2]

PROFITS : [13, 12, 20, 15, 18]
WEIGHTS : [1, 3, 5, 5, 9]

item 3, item 1, item 5, item 4, item 2

Weight Capacity of Bag : 10
We must find the maximum profit for the weight capacity of the bag, by fitting in items into the bag.



- ITER 1 :
  item 3 : Profit : 13
  weight : 1 (fits fully in bag)
  New weight capacity = 9
  Profit of bag = 13

- ITER 2 :
  item 1 : Profit : 12
  weight : 3 (fits fully in bag)
  New weight capacity = 9 - 3 = 6
  Profit of bag = 13 + 12 = 25

- ITER 3 :
  item 5 : Profit : 20
  weight : 5 (fits fully in bag)
  New weight capacity = 6 - 5 = 1
  Profit of bag = 25 + 20 = 45

- ITER 4 :
  item 4 : Profit : 15
  weight : 5 (Does NOT fit fully in bag)
  New weight capacity = 1 - 1 = 0 $\rightarrow$ (BAG FULL)
  Profit of bag = 45 + $\left[\text{profit of } \frac{1}{5} \text{th of item 4} \right]$

  = 45 + 3
  = 48

Stop execution as bag is now full
Final Answer : 48 is the profit of the full bag

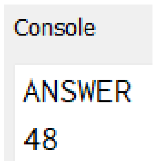
Code:

```
# we have used the same problem as solved manually above

.data
op: .string ANSWER\n
profit: .word 12,18,13,15,20
weights: .word 3,9,1,5,5
ppw: .word 0,0,0,0,0
rm: .word 0,0,0,0,0
number: .word 5
capacity: .word 10
.text
la x1,profit
la x2,weights
la x3,ppw
la x15,rm
la x4, number
lw x5,0(x4)
comeback:
beqz x5,moveon
lw x6,0(x1)
lw x7,0(x2)
rem x26,x6,x7
div x8,x6,x7
sw x8,0(x3)
sw x26,0(x15)
addi x1,x1,4
addi x2,x2,4
addi x3,x3,4
addi x15,x15,4
addi x5,x5,-1
j comeback


moveon:

la x1,ppw
la x29,profit
la x30,profit
la x21,weights
la x22,weights
la x9,ppw
la x15,rm
la x17,rm
la x2,number
la x25,capacity

lw x3,0(x2)
add x7,x3,x0
addi x3,x3,-1
beqz x3,the_start

back:
```

```
lw x4,0(x1)
lw x23,0(x21)
lw x27,0(x29)
lw x16, 0(x15)
lw x5,4(x1)
lw x24,4(x21)
lw x28,4(x29)
lw x26,4(x15)
blt x5,x4,skip
beq x5,x4,eskip
sw x5,0(x1)
sw x24,0(x21)
sw x28,0(x29)
sw x26,0(x15)
sw x4,4(x1)
sw x23,4(x21)
sw x27,4(x29)
sw x26,4(x15)

eskip: bge x16,x26,eeskip
sw x5,0(x1)
sw x24,0(x21)
sw x28,0(x29)
sw x4,4(x1)
sw x23,4(x21)
sw x27,4(x29)
eeskip:
skip: addi x3,x3,-1
addi x1,x1,4
addi x21,x21,4
addi x29,x29,4
addi x15,x15,4
bnez x3,back
add x1,x9,x0
add x21,x22,x0
add x29,x30,x0
add x15,x17,x0
addi x7,x7,-1
beqz x7,the_start
add x3,x7,x0
j back

the_start:
la x2,number
lw x20, 0(x2)
lw x21, 0(x25)
la x1, profit # Set up pointers to the arrays
la x2, weights
li x3, 0 # Loop Counter
li x4, 0 # Current capacity
li x11, 0 # Initialize the result (knapsack value) to 0

knapsack_loop:
```

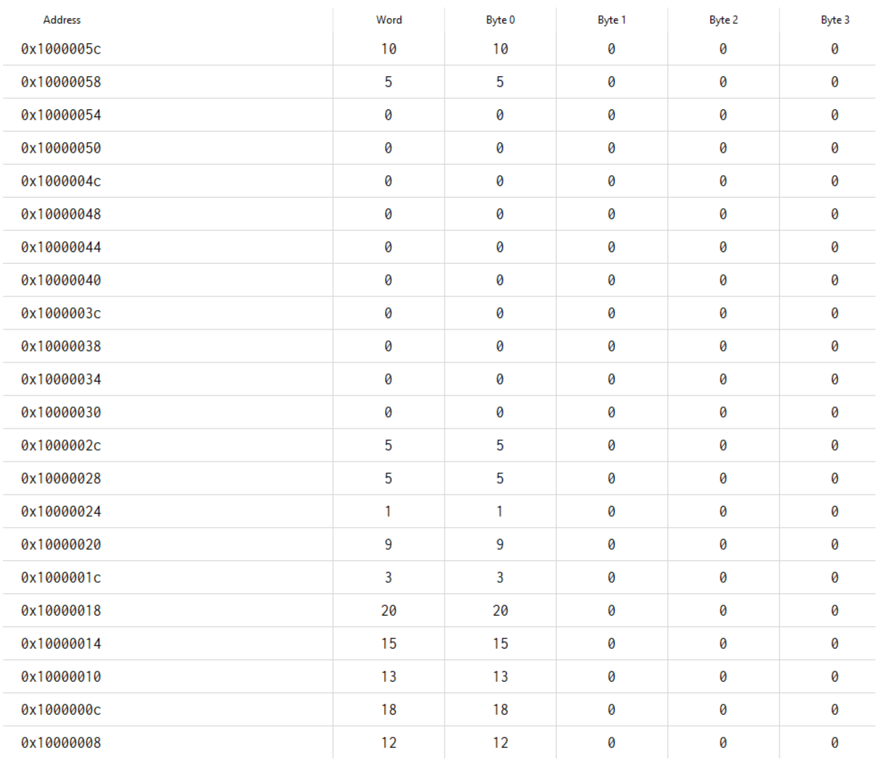
```
bge x3, x20, knapsack_end # If the counter reaches the end of the arrays, branch to knapsack_end
lw x5, 0(x1) # Load value of current item
lw x6, 0(x2) # Load weight of current item
sub x7, x21, x4 # Calculate remaining capacity in the bag
bge x7,x0,item_fits # If the current item can fit in the bag, branch to item_fits
j next_iteration # If item doesn't fit, branch to the next_iteration label

item_fits:
sub x8, x7, x6 # Calculate the remaining capacity after adding the item
bge x8,x0,looper1
mul x27,x5,x7
div x27,x27,x6
add x11,x11,x27
j skipper
looper1:
add x11, x11, x5 # Update the result (knapsack value)
skipper: add x4, x4, x6

next_iteration: # To move to the next item
addi x1, x1, 4 # Increment value pointer
addi x2, x2, 4 # Increment weight pointer
addi x3, x3, 1 # Increment loop counter
j knapsack_loop # Repeat the loop

knapsack_end:
la a0, op # Load address of string 'ANSWER\n' to a0
li a7, 4 # Load the system call number for printing strings into a7
ecall # Make the system call to print the string
mv a0, x11 # Load result (Knapsack value) to a0
li a7,1 # Load the system call number for printing integers into a7
ecall # Make the system call to print the integer
nop
```

Output:



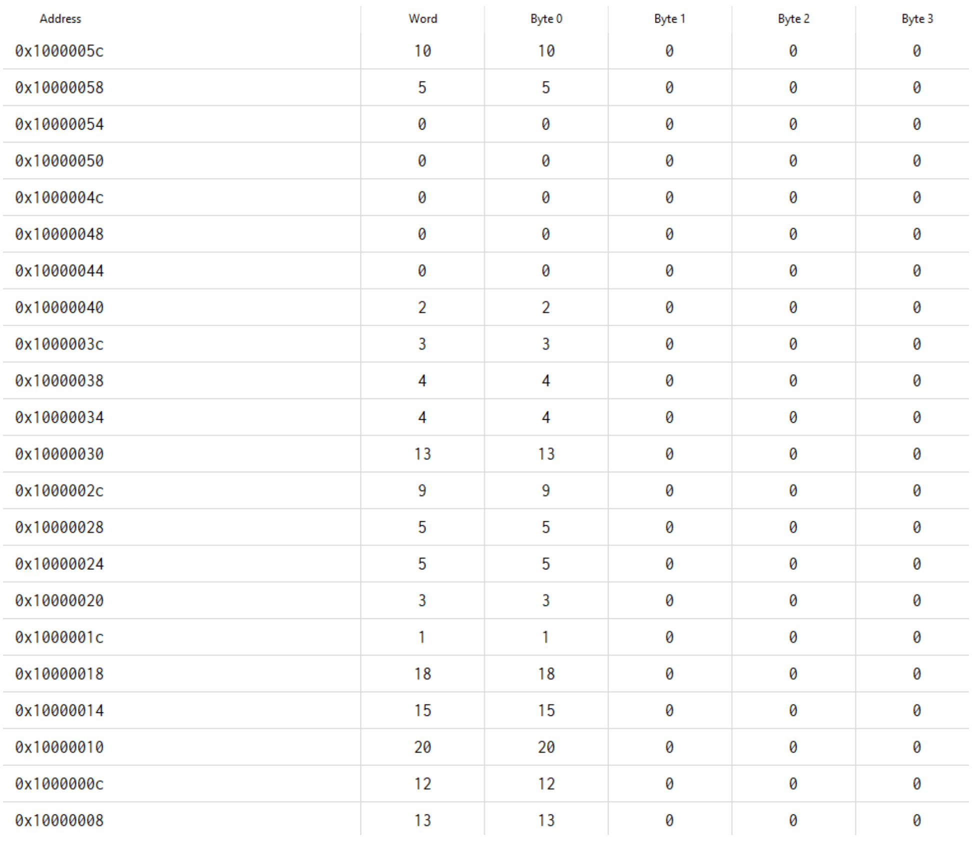
```
ANSWER
48
```

Memory Before Execution:



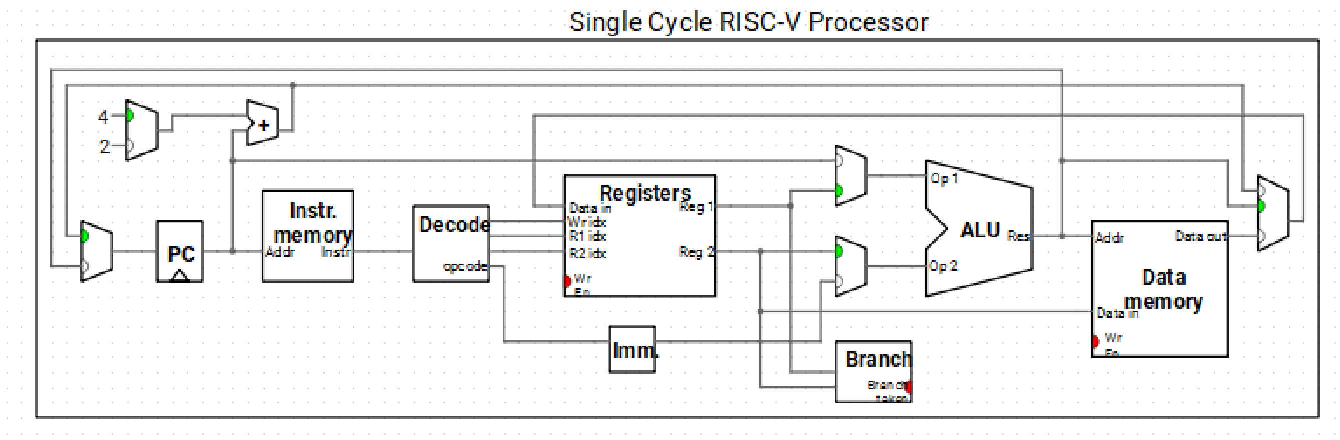
Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x1000005c	10	10	0	0	0
0x10000058	5	5	0	0	0
0x10000054	0	0	0	0	0
0x10000050	0	0	0	0	0
0x1000004c	0	0	0	0	0
0x10000048	0	0	0	0	0
0x10000044	0	0	0	0	0
0x10000040	0	0	0	0	0
0x1000003c	0	0	0	0	0
0x10000038	0	0	0	0	0
0x10000034	0	0	0	0	0
0x10000030	0	0	0	0	0
0x1000002c	5	5	0	0	0
0x10000028	5	5	0	0	0
0x10000024	1	1	0	0	0
0x10000020	9	9	0	0	0
0x1000001c	3	3	0	0	0
0x10000018	20	20	0	0	0
0x10000014	15	15	0	0	0
0x10000010	13	13	0	0	0
0x1000000c	18	18	0	0	0
0x10000008	12	12	0	0	0

Memory After Execution:

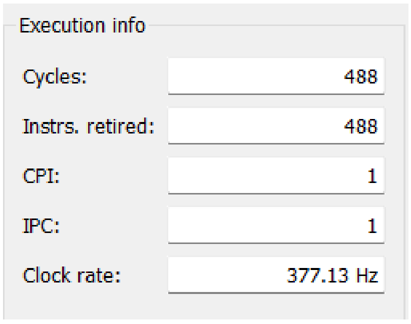


Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x1000005c	10	10	0	0	0
0x10000058	5	5	0	0	0
0x10000054	0	0	0	0	0
0x10000050	0	0	0	0	0
0x1000004c	0	0	0	0	0
0x10000048	0	0	0	0	0
0x10000044	0	0	0	0	0
0x10000040	2	2	0	0	0
0x1000003c	3	3	0	0	0
0x10000038	4	4	0	0	0
0x10000034	4	4	0	0	0
0x10000030	13	13	0	0	0
0x1000002c	9	9	0	0	0
0x10000028	5	5	0	0	0
0x10000024	5	5	0	0	0
0x10000020	3	3	0	0	0
0x1000001c	1	1	0	0	0
0x10000018	18	18	0	0	0
0x10000014	15	15	0	0	0
0x10000010	20	20	0	0	0
0x1000000c	12	12	0	0	0
0x10000008	13	13	0	0	0

Processor Used



Execution Information:


Execution info

Cycles:	488
Instrs. retired:	488
CPI:	1
IPC:	1
Clock rate:	377.13 Hz

Conclusion:

We have succeeded in a simple implementation of the solution for the Fractional Knapsack problem. This can serve as a stepping stone for implementing more complex approaches at the assembly level such as recursion and dynamic programming, which can give accurate results for very large sets of data.