Computer Organization and Architecture

Name: Pallav Biyala
Roll No.: 24110234

Course: Computer Organization and Architecture
Date: 13th February, 2025

Assignment – II

Q1 Write a function in MIPS32 assembly that computes the Highest value & index, Lowest value & index and Average value of an Array arr [100]. Assume the base address of Array arr is available at $gp. The results will be stored at $v0, $v1, $s0, $s1 and Ss2.

Sol 1: So before going to MIPS Assembly, lets just first write the code in high level language. So here we can choose either C or C++. For convenience, we will stick to C++.

The high level code for the following function would be:

```
vector<int> find(vector<int> &arr, int n){
    int max_idx = 0;
    int max = arr[0];
    int min_idx = 0;
    int min = arr[0];
    int sum = 0;

    for (int i = 0; i<n; ++i){
        // finding max
        if (arr[i]>max){
            max = arr[i];
            max_idx = i;
        }

        // finding min
        if (min>arr[i]){
            min = arr[i];
            min_idx = i;
        }

        // finding sum
        sum += arr[i];
    }

    vector <int> result = {max,max_idx,min,min_idx,sum / n};
    return result;
}
```

Hence, lets convert it to assembly. We are given:

- arr is available at $gp.
- The results will be stored at $v0, $v1, $s0, $s1 and $s2.
- So let's store:
  - max → $v0
  - min → $v1
  - max_idx → $s0

- min_idx → $s1
- average → $s2

So the assembly code would be:

Assembly Code:

```
.text
.globl main

main:
        move $a0, $gp #since it is given arr[0] base address is at gp so lets store address in a0
        li $a1, 100 #since size is given as 100
        jal find
        li $v0,10
        syscall

# Function call
find:
        # since we have to store output in callee saved registers, we are avoiding to store it in stack as we need data to be in them
        lw $t0, 0($a0) # max = a[0]
        lw $t1, 0($a0) # min = a[0]
        li $s0, 0 # max_idx = 0
        li $s1, 0 # min_idx = 0
        li $s2, 0 # sum = 0

        move $t2, $a1 # $t2 = 100
        move $t3, $a0 # t3 = address of arr[0]
        addi $t4, $0, 0 # i = 0
```

```
        # finding all parameters
        Loop:
                beq $t2,$0, end # if t2 == 0, we are done. skip return
                lw $t5, 0($t3) # now t5 = arr[i]
                # max time
                ble $t5, $t0, min # if t5<=max branch to min as max is not here
                        move $t0, $t5 # max = t5
                        move $s0, $t4 # max_idx = t4 = i

                # min time
                min:
                bge $t5, $t1, sum # if t5>=min branch
                        addi $t1, $t5, 0 # min = t5
                        addi $s1, $t4, 0 # min_idx = t4 = i

                sum:
                # sum time
                add $s2, $s2, $t5 # s2 += t5

                # incrementing i and moving to next element
                addi $t4, $t4, 1 # i++
                addi $t3, $t3, 4 # moving to next element
                addi $t2, $t2, -1 # n--
                j Loop
```

```
        end:
                move $v0, $t0
                move $v1, $t1

                # max and min donw just average left
                move $t0, $a1
                div $s2, $t0
                mflo $s2
                jr $ra
```

Q2 MIPS program to add two matrices A and B of size N x N and store the result in C, where N is 4. You may please check on how to initialize the matrix on a given simulator.

Sol. 2: The C++ code to add two matrices A and B is:

C++ Code:

```
#include <iostream>
using namespace std;
```

```cpp
vector<vector<int>> sum_matrix(vector<vector<int>> &A, vector<vector<int>>&B){
    int n = A.size();
    vector<vector<int>> sum(n, vector<int> (n,0));

    for (int i = 0; i<n; ++i){
        for (int j = 0; j<n;++j){
            sum[i][j] = A[i][j]+B[i][j];
        }
    }
    return sum;
}

int main()
{

    return 0;
}
```

So to access each element address, we would have only base address of A[0][0] or B[0][0]. So to simplify logic, we know memory is just a contiguous block of memory. Hence, we can consider a 2D matrix as a flattened array. Hence lets say:

Assumptions:

- Size of matrix is 3 * 3, hence 9 elements so 36 bytes storage needed.
- Base address of A is being pointed by $gp.
- Base address of B is at 40($gp)
- Base address of result would be at 80($gp)
- N is stored at 120($gp)

So we can maintain a register for A, for B and for result and every time move it by 4 bytes. By this way, we could keep track of all corresponding elements and we won't require external logic too!

Now we can easily write assembly code:

Assembly:

```
        .data
A:      .word 1,2,3,4,5,6,7,8,9
B:      .word 9,8,7,6,5,4,3,2,1
result: .space 36       # 9 elements * 4 bytes
N:      .word 3

        .text
        .globl main

main:
        # Adding elements to the arguement registers
        la $a0, A
        la $a1, B
        la $a2, result
        lw $a3, N

        jal sum_matrix
        li $v0,10
        syscall

# Function call
sum_matrix:
        # Instead of two for loops since we are flattening matrices, we can just move till n*n elements
        move $t1,$a3 # t1 = n
        mult $t1,$t1
        mflo $t2 # $t2 = n*n
```

```
    Loop:
            beq $t2, $0, end # if n*n == 0, we have done iteration n*n times and covered all n elements
            lw $t3, 0($a0) # $t3 = A[0]
            lw $t4, 0($a1) # $t4 = B[0]

            add $t5, $t3,$t4 # t5 = A[0] + B[0]
            sw $t5, 0($a2) # result[0] = $t5

            # moving to next elements
            addi $a0, $a0, 4
            addi $a1, $a1, 4
            addi $a2, $a2, 4

            # n --
            addi $t2, $t2, -1

            j Loop
    end:
            jr $ra
```

Hence, here to simplify problem we considered the fact that we have N*N matrices and memory is contagious block of elements, and hence we can flatten the matrices and then simply add them.

Q3 Implement a quick sort in MIPS.

Sol. 3: The C++ code for the quick sort is:

```cpp
class quicksort{
private:
    void swap(int &a, int &b){
        int temp = a;
        a = b;
        b = temp;
    }

    int partition(vector<int>&arr, int low, int high){
        int pivot = arr[high]; // taking pivot as high element
        int i = low-1;

        for (int j = low; j< high; ++j){
            if (arr[j] <= pivot){
                i++;
                swap(arr[i],arr[j]);
            }
        }
        swap(arr[i+1], arr[high]);
        return i+1;
    }

    void quicksort(vector<int>&arr, int low, int high){
        if (low<high){
            int p = partition(arr,low,high); // p is at correct position now
            quicksort(arr, low, p-1); // sort left half
            quicksort(arr, p+1, high); // sort right half
        }
    }
```

```
public:
      vector<int> sort(vector<int>&arr){
        int n = arr.size();
        quicksort(arr, 0, n-1);
      }
};
```

Assembly:

```
        .data
arr     .word 9,1,5,2,10,12,6
n       .word 7

        .text
        .globl main
```

```
# Sort function is non leaf
sort:
        addi $sp, $sp, -4
        sw $ra, 0($sp)

        # $a0 already contains arr
        li $a1, 0 # a1 = 0 = low
        lw $t0, n # t0 = n = high
        addi $a2, $t0, -1 # a2 = high-1 = n-1
        jal quicksort

        lw $ra, 0($sp)
        addi $sp, $sp, 4
        jr $ra
```

```
# Swap function is leaf function so no need to store $ra.
swap:
        lw $t8, 0($a0) # $t0 = arr[a]
        lw $t9, 0($a1) # $t1 = arr[b]

        sw $t9, 0($a0) # storing value of arr[b] in arr[a]
        sw $t8, 0($a1) # storing value of arr[a] in arr[b]
        jr $ra
```

```
# Partition function is non leaf function need to store $ra
partition:
        # Function prologue
        addi $sp, $sp, -4
        sw $ra, 0($sp)

        # Function Body
        move $t0, $a0 # t0 stores address of arr[0]
        move $t1, $a1 # stores LOW INDEX
        move $t2, $a2 # stores HIGH INDEX

        # now we need to store arr[high] for pivot: so address = high*4 + base
        sll $t3, $t2, 2
        add $t3, $t3, $t0 # $t3 stores address of arr[high]
        lw $t3, 0($t3) # $t3 = pivot = arr[high]

        # i and j pointers
        addi $t4, $t1,-1 # i = low -1
        move $t5,$t1  # j = low
```

```
        # Now loop time
        Loop:
                bge $t5, $t2, skip # if j>=high skip

                # store arr[j] = j*4 + base
                sll $t6, $t5, 2
                add $t6, $t6, $t0 # address of arr[j]
                lw $t8, 0($t6) # arr[j]

                # now if condition
                bge $t8, $t3, else
                addi $t4,$t4,1 # i++

                # Now we need to load arr[i] and arr[j] in a0 and a1. since t6 stores arr[j] it moves to a1
                # finding arr[i] = i*4 +base
                sll $t7, $t4, 2
                add $t7, $t7, $t0 # $t7 stores address of arr[i]
                move $a0, $t7
                move $a1, $t6

                # now call swap
                jal swap
                else:
                        addi $t5, $t5, 1
                        j Loop


        skip:
                # now we just need to swap arr[i+1] and arr[high]
                # so address = (i+1)*4 + base
                addi $t4, $t4, 1 # i+1
                sll $t7, $t4, 2
                add $t7, $t7, $t0 # $t7 stores address of arr[i+1]
                move $a0, $t7

                # so address = (HIGH)*4 + base
                sll $t8, $t2, 2
                add $t8, $t8, $t0 # address of arr[high]
                move $a1, $t8

                jal swap

                move $v0, $t4

        # Function epilogue
        lw $ra, 0($sp)
        addi $sp,$sp,4
        jr $ra
```

```
# Quicksort function is a non leaf function
quicksort:
        # Prologue
        addi $sp, $sp, -20
        sw $ra, 0($sp)
        sw $t0, 4($sp)
        sw $t1, 8($sp)
        sw $t2, 12($sp)
        sw $t3, 16($sp)


        # Function Body
        move $t0, $a0 # address of arr[0]
        move $t1, $a1 # address of low
        move $t2, $a2 # address of high

        bge $t1,$t2, end # if low>= high skip
        # partition uses same $a0,$a1,$a2
        move $a0, $t0
        move $a1, $t1
        move $a2, $t2
        jal partition
        move $t3, $v0 # t3 = p
```

```
        # calling quicksort low,p-1
        move $a0, $t0
        move $a1, $t1
        addi $a2, $t3, -1
        jal quicksort
        lw $t3, 16($sp)    # restoring pivot index


        # calling quicksort p+1,high
        move $a0, $t0
        addi $a1, $t3,1
        move $a2, $t2
        jal quicksort


        # Function epilogue
        end:
                lw $ra, 0($sp)
                lw $t0, 4($sp)
                lw $t1, 8($sp)
                lw $t2, 12($sp)
                lw $t3, 16($sp)
                addi $sp, $sp, 20
                jr $ra
```

Q4 Write a MIPS program to perform a binary search on an existing and non-existing element a sorted Array A [10]

Sol. 4: The C++ code for binary code is:

```
bool binary_search(vector<int>&arr, int n, int k){
    int low = 0;
    int high = n - 1;

    while (low<= high){
        int mid = low + (high - low)/2;
        if (arr[mid] == k){
            return true;
        }

        else if (arr[mid]>k){
            // search left
            high = mid - 1;
        }

        else{
            // search right
            low = mid+1;
        }
    }
    return false;
}
```

Assembly Code:

```
.data
arr:    .word 2,5,10,11,13,15,21
n:      .word 7
key:    .word 21

.text
.globl main

main:
        la   $a0, arr         # base address of array
        lw   $a1, n           # n
        lw   $a2, key         # key to search
        jal  binary_search

        li   $v0, 10
        syscall
```

```
# Function call
binary_search:
        li   $t0, 0           # low = 0
        addi $t1, $a1, -1     # high = n-1
        move $t4, $a2         # key

        Loop:
                bgt  $t0, $t1, not_found   # if low > high → exit

                # mid = (low + high)/2
                add  $t2, $t0, $t1
                srl  $t2, $t2, 1

                # load arr[mid], $t2 contains index and hence address = base + index*4
                sll  $t3, $t2, 2      # mid * 4
                add  $t3, $t3, $a0    # address = base + offset
                lw   $t3, 0($t3)      # arr[mid]

                # if arr[mid] == key
                beq  $t3, $t4, found

                # if arr[mid] > key
                bgt  $t3, $t4, go_left


                # else arr[mid] < key
                addi $t0, $t2, 1      # low = mid + 1
                j Loop

        go_left:
                addi $t1, $t2, -1     # high = mid - 1
                j Loop

        found:
                li $v0, 1 # $v0 = 1 means we found the element (true)
                jr $ra

        not_found:
                li $v0, 0 # $v0 = 0 means not found (false)
                jr $ra
```

Q5 Write a MIPS program to multiply square matrices of size N x N where N is 5. Compare the code you wrote w.r.t the earlier version of the code (in Assignment 1) generated by the compiler when N is set to 5. What differences do you observe?

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      C[i][j] += A[i][k] * B[k][j];
```

Sol. 5: The C++ code for this question is:

```
vector<vector<int>> matrix_multiplication(vector<vector<int>> &A, vector<vector<int>> &B, int n){
```

```
    vector<vector<int>> result (n, vector<int> (n,0));

    for (int i = 0; i<n;++i){
        for (int j = 0; j<n;++j){
            for (int k = 0; k<n; ++k){
                result[i][j] += A[i][k]*B[k][j];
            }
        }
    }

    return result;
}
```

Now the main and hard thing here is to keep track of indices when we consider matrices as flattened. So now suppose we have a matrix:

$$A = \begin{bmatrix} 1 & 2 & 5 \\ 8 & 4 & 5 \\ 7 & 0 & 1 \end{bmatrix}$$

Now when we flatten it:

$$A = [\ 1\ 2\ 5\ 8\ 4\ 5\ 7\ 0\ 1]$$

So now let's see how indices changed:

- [0][0] → 0
- [0][1] → 1
- [0][2] → 2
- [1][0] → 3
- [1][1] → 4
- [1][2] → 5
- [2][0] → 6
- [2][1] → 7
- [2][2] → 8

So now for if i denotes row and j denotes column:

- For i = 0, index = j
- For i = 1, index = j + 3 (and basically n = 3 → n + j)
- For i = 2, index = j + 6 = j + 2*3 (and this means index = n*i + j as here i was 2 and it works for all others)

Hence in general to find index of a matrix in a 2D array when flattened:

$$\text{Index} = n*i + j$$

And we know to know address of index in **Memory:**

Address = base address of arr[0] + index*4

Which gives:

$$\text{Address} = \text{base} + (n*i + j)*4$$

Hence we will need to play a lot with this equation!

Now, here:

We have n = 5, hence formula becomes:

Address = base + (5i + j)*4

So lets firstly map registers for convenience:

- i → $s1
- j → $s2
- k → $s3
- n → $s4
- $a0, $a1 → Matrices base address of A,B
- $s0 → Base address of Result
- $t5 → sum
- $t1 → A[i][k]
- $t3 → B[k][j]

Rest all we can use for address calculation etc purposes, so even if those overwritten we don't care.

Assembly Code:

```
.data
A       .word 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,0,2,2,3,2,5
B       .word 1,2,3,0,5,6,7,8,1,10,1,2,0,4,5,6,9,8,3,0,1,6,9,2,5
result  .space 100

.text
.globl main

main:
        la $a0, A # $a0 stores address of A
        la $a1, B # $a1 stores address of B
        la $a2, result # $a2 stores address of result
        jal matrix_multiplication

        li $v0, 10
        syscall

# Function call
matrix_multiplication:
        # Function prologue
        addi $sp, $sp, -20
        sw $s0, 0($sp)
        sw $s1, 4($sp)
        sw $s2, 8($sp)
        sw $s3, 12($sp)
        sw $s4, 16($sp)


        move $s0, $a2 # stores base address of result

        # Creating variables
        li $s1, 0 #i
        li $s4, 5 #n

        for_1:
                beq $s1, $s4, end # if i == n we are done branch off
                li $s2, 0 #j

                for_2:
                        beq $s2, $s4, i_plus_plus # if j == n means i = current value is completed now we need to incremenet i
                        li $t5, 0 # sum container
                        li $s3, 0 #k

                        for_3:
                                beq $s3, $s4, j_plus_plus # if k == n means j = current value is done now we need to increment j

                                # Finding A[i][k]
                                # Step 1: Find address = base + (i*5 + k)*4
                                mult $s4, $s1
                                mflo $t0 # let t0 store address
                                add $t0, $t0, $s3 # 5i+k done
                                sll $t0, $t0, 2 # (5i+k)*4 done
                                add $t0, $t0, $a0 # base + (5i+k)*4 computed
```

```
                        # Now extract value
                        lw $t1, 0($t0) # $t1 stores A[i][k]

                        # Finding B[k][j]
                        # Step 1: Find address = base + (k*5 + j)*4
                        mult $s4, $s3
                        mflo $t2 # let t2 store address
                        add $t2, $t2, $s2 # 5k+j done
                        sll $t2, $t2, 2 # (5k+j)*4 done
                        add $t2, $t2, $a1 # base + (5k+j)*4 computed

                        # Now extract value
                        lw $t3, 0($t2) # $t3 stores B[k][j]

                        # Let $t5 sontain sum
                        mult $t1, $t3
                        mflo $t6
                        add $t5, $t5, $t6 # sum += A[i][k]*B[k][j]

                        # Now go for next k++
                        addi $s3, $s3, 1 # k++
                        j for_3


                j_plus_plus:
                        # storing value of $t5 in result
                        # Finding result[i][j] to know where to store

                        # Step 1: Find address = base + (i*5 + j)*4
                        mult $s4, $s1
                        mflo $t4 # let t0 store address
                        add $t4, $t4, $s2 # 5i+j done
                        sll $t4, $t4, 2 # (5i+j)*4 done
                        add $t4, $t4, $s0 # base + (5i+j)*4 computed

                        # Now we can store result in $t4 as $t4 contains address
                        sw $t5, 0($t4)

                        # now j++
                        addi $s2, $s2, 1 #j++
                        j for_2
        i_plus_plus:
                addi $s1, $s1, 1 #i++
                j for_1

end:
        lw $s0, 0($sp)
        lw $s1, 4($sp)
        lw $s2, 8($sp)
        lw $s3, 12($sp)
        lw $s4, 16($sp)
        addi $sp, $sp, 20

        # adding returning value
        move $v0, $a2 # as $s0 stores base address of result
        jr $ra
```

→ Comparison with Earlier Code generated by compiler:

So, if we compare our handwritten code with the compiler generated for same problem, we found out that:

1. The compiler has written a huge amount of code with lines crossing 200 – 300 (no optimized across 400) lines of instructions. On the other hand, we used minimal 100 lines of codes with lots of spaces.
2. Compiler has excessively used stack pointers, frame pointers and saved too many registers. While we ignored frame pointer and use stack pointer only to store callee saved registers.
3. Compiler uses stack for loop variables while we did all loop in temporary registers, making sure that if value is needed, we don't store it in it again

Hence, the compiler-generated code is significantly longer and uses more stack space compared to our assembly code.

The compiler saves additional registers, uses a frame pointer, and often stores loop variables in memory. Our implementation though keeps loop variables in registers and performs direct index calculations, making it shorter and more readable. The compiler-generated code is more general and adheres strictly to calling conventions, while the ours is optimized for clarity and a fixed matrix size and easier to understand.

Q6 Assemble the following instructions based on the MIPS data-card shared on MS teams. Assume the string "DEADEND0" is stored at the base address pointed by $gp.

```
Do: addi $a0, $0, $0
    lw  $t1,  0($gp)
    lw  $t2,  4($gp)
    add $t3, $t2, $t1
    sub $t4, $t2, $t1
    slt $t5, $t4, $t3
    beq $t5, $0, noswp
    nop
swp: sw  $t2  0($gp)
     sw  $t1  4($gp)
noswp j do
```

Sol. 6: So in order to write correct machine code, let's allocate addresses to each instruction:

Address	Instruction
0x00400000	addi $a0, $0, 0
0x00400004	lw $t1, 0($gp)
0x00400008	lw $t2, 4($gp)
0x0040000C	add $t3, $t2, $t1
0x00400010	sub $t4, $t2, $t1
0x00400014	slt $t5, $t4, $t3
0x00400018	beq $t5, $0, noswp
0x0040001C	nop
0x00400020	sw $t2 0($gp)
0x00400024	sw $t1 4($gp)
0x00400028	j Do

So, to refer how machine codes were formed, we can refer the following notes from notepad (a text file is also attached in the github)

```
1. addi $a0, $0, 0

Syntax: addi $rt, $rs, imm
Here,
        Type = I Type [op rs rt imm]
        addi --> op code = 8 -> 001000 (6 bits)
        rt = $a0 --> 4 -> 00100 (5 bits)
        rs = $0 --> 0 -> 00000 (5 bits)
        imm = 0 --> 0 --> 0000000000000000 (16 bits)

Hence machine code becomes: 001000 00000 00100 0000000000000000 => 0x20040000

2. lw  $t1,  0($gp)

Syntax: lw $rt, imm($rs)
Here,
        Type = I Type [op rs rt imm]
        lw --> op code = 35 -> 100011 (6 bits)
        rt = $t1 --> 9 -> 01001 (5 bits)
        rs = $gp --> 28 -> 11100 (5 bits)
        imm = 0 --> 0 --> 0000000000000000 (16 bits)

Hence machine code becomes: 100011 11100 01001 0000000000000000 => 0x8F890000

3. lw  $t2,  4($gp)

Syntax: lw $rt, imm($rs)
Here,
        Type = I Type [op rs rt imm]
        lw --> op code = 35 -> 100011 (6 bits)
        rt = $t2 --> 10 -> 01010 (5 bits)
        rs = $gp --> 28 -> 11100 (5 bits)
        imm = 4 --> 4 --> 0000000000000100 (16 bits)

Hence machine code becomes: 100011 11100 01010 0000000000000100 => 0x8F8A0004
```

```
4. add $t3, $t2, $t1

Syntax: add $rd, $rs, $rt
Here,
        Type = R Type [op rs rt rd shamt funct]
        add --> op code = 0 -> 000000 (6 bits)
        rd = $t3 --> 11 -> 01011 (5 bits)
        rs = $t2 --> 10 -> 01010 (5 bits)
        rt = $t1 --> 9 -> 01001 (5 bits)
        shamt = 0 --> 0 -> 00000 (5 bits)
        funct = 32 --> 32 -> 100000 (6 bits)

Hence machine code becomes:  000000 01010 01001 01011 00000 100000 => 0x01495820

5. sub $t4, $t2, $t1

Syntax: sub $rd, $rs, $rt
Here,
        Type = R Type [op rs rt rd shamt funct]
        sub --> op code = 0 -> 000000 (6 bits)
        rd = $t4 --> 12 -> 01100 (5 bits)
        rs = $t2 --> 10 -> 01010 (5 bits)
        rt = $t1 --> 9 -> 01001 (5 bits)
        shamt = 0 --> 0 -> 00000 (5 bits)
        funct = 34 --> 32 -> 100010 (6 bits)

Hence machine code becomes:  000000 01010 01001 01100 00000 100010 => 0x01496022
```

```
6. slt $t5, $t4, $t3

Syntax: slt $rd, $rs, $rt
Here,
        Type = R Type [op rs rt rd shamt funct]
        slt --> op code = 0 -> 000000 (6 bits)
        rd = $t5 --> 13 -> 01101 (5 bits)
        rs = $t4 --> 12 -> 01100 (5 bits)
        rt = $t3 --> 11 -> 01011 (5 bits)
        shamt = 0 --> 0 -> 00000 (5 bits)
        funct = 42 --> 42 -> 101010 (6 bits)

Hence machine code becomes:  000000 01100 01011 01101 00000 101010 => 0x018B682A

7. beq $t5, $0, noswp

Syntax: beq $rs, $rt, imm
Here,
        Type = I Type [op rs rt imm]
        beq --> op code = 4 -> 000100 (6 bits)
        rs = $t5 --> 13 -> 01101 (5 bits)
        rt = $0 --> 0 -> 00000 (5 bits)
        imm = 3 --> 3 --> 0000000000000011 (16 bits) # as noswp is after three instructions

Hence machine code becomes: 000100 01101 00000 0000000000000011 => 0x11A00003

8. nop # this means sll $0, $0, 0

Syntax: sll $rd, $rt, shamt
Here,
        Type = R Type [op rs rt rd shamt funct]
        slt --> op code = 0 -> 000000 (6 bits)
        rd = $0 --> 0 -> 00000 (5 bits)
        rs = 0 --> 0 -> 00000 (5 bits) # as its unused
        rt = $0 --> 0 -> 00000 (5 bits)
        shamt = 0 --> 0 -> 00000 (5 bits)
        funct = 0 --> 0 -> 000000 (6 bits)

Hence machine code becomes:  000000 00000 00000 00000 00000 000000 => 0x00000000
```

```
9. sw  $t2  0($gp)

Syntax: sw $rt, imm($rs)
Here,
        Type = I Type [op rs rt imm]
        sw --> op code = 43 -> 101011 (6 bits)
        rt = $t2 --> 10 -> 01010 (5 bits)
        rs = $gp --> 28 -> 11100 (5 bits)
        imm = 0 --> 0 --> 0000000000000000 (16 bits)

Hence machine code becomes: 101011 11100 01010 0000000000000000 => 0xAF8A0000

10. sw  $t1  4($gp)

Syntax: sw $rt, imm($rs)
Here,
        Type = I Type [op rs rt imm]
        sw --> op code = 43 -> 101011 (6 bits)
        rt = $t1 --> 9 -> 01001 (5 bits)
        rs = $gp --> 28 -> 11100 (5 bits)
        imm = 4 --> 4 --> 0000000000000100 (16 bits)

Hence machine code becomes: 101011 11100 01001 0000000000000100 => 0xAF890004
```

```
11. j do

Syntax: j target address
Here,
        Type = J Type [op target address]
        j --> op code = 2 -> 000010 (6 bits)
        target address -->

        Now, to find the target address of j, we would need 26 bits of address of Do, which is: 0x00400000.
        So, now we know:
                        ETA = {(PC+4[31:28]), target address<<2}

        Hence if we reverse engineer, we can just do right shift instead of left and then take lower 26 bits:
                        0x00400000 >> 2 = 0x00100000
        In binary:
                        0x00100000 = 0000 0000 0001 0000 0000 0000 0000 0000

        And the lower 26 bits are:
                        Target address = 00 0001 0000 0000 0000 0000 0000

Hence now we can form machine code: 000010 00 0001 0000 0000 0000 0000 0000 => 0x08100000
```

The machine code for the following program in assembly would be:

→ **Binary Form:**

0x00400000: 001000 00000 00100 0000000000000000

0x00400004: 100011 11100 01001 0000000000000000

0x00400008: 100011 11100 01010 0000000000000100

0x0040000C: 000000 01010 01001 01011 00000 100000

0x00400010: 000000 01010 01001 01100 00000 100010

0x00400014: 000000 01100 01011 01101 00000 101010

0x00400018: 000100 01101 00000 0000000000000011

0x0040001C: 000000 00000 00000 00000 00000 000000

0x00400020: 101011 11100 01010 0000000000000000

0x00400024: 101011 11100 01001 0000000000000100

0x00400028: 000010 00000100000000000000000000

→ **Hexadecimal Form:**

0x00400000: 0x20040000

0x00400004: 0x8F890000

0x00400008: 0x8F8A0004

0x0040000C: 0x01495820

0x00400010: 0x01496022

0x00400014: 0x018B682A

0x00400018: 0x11A00003

0x0040001C: 0x00000000

0x00400020: 0xAF8A0000

0x00400024: 0xAF890004

0x00400028: 0x08100000