STACK-BASED ISA

CSC-201: Computer Architecture and Organization

End Term Project

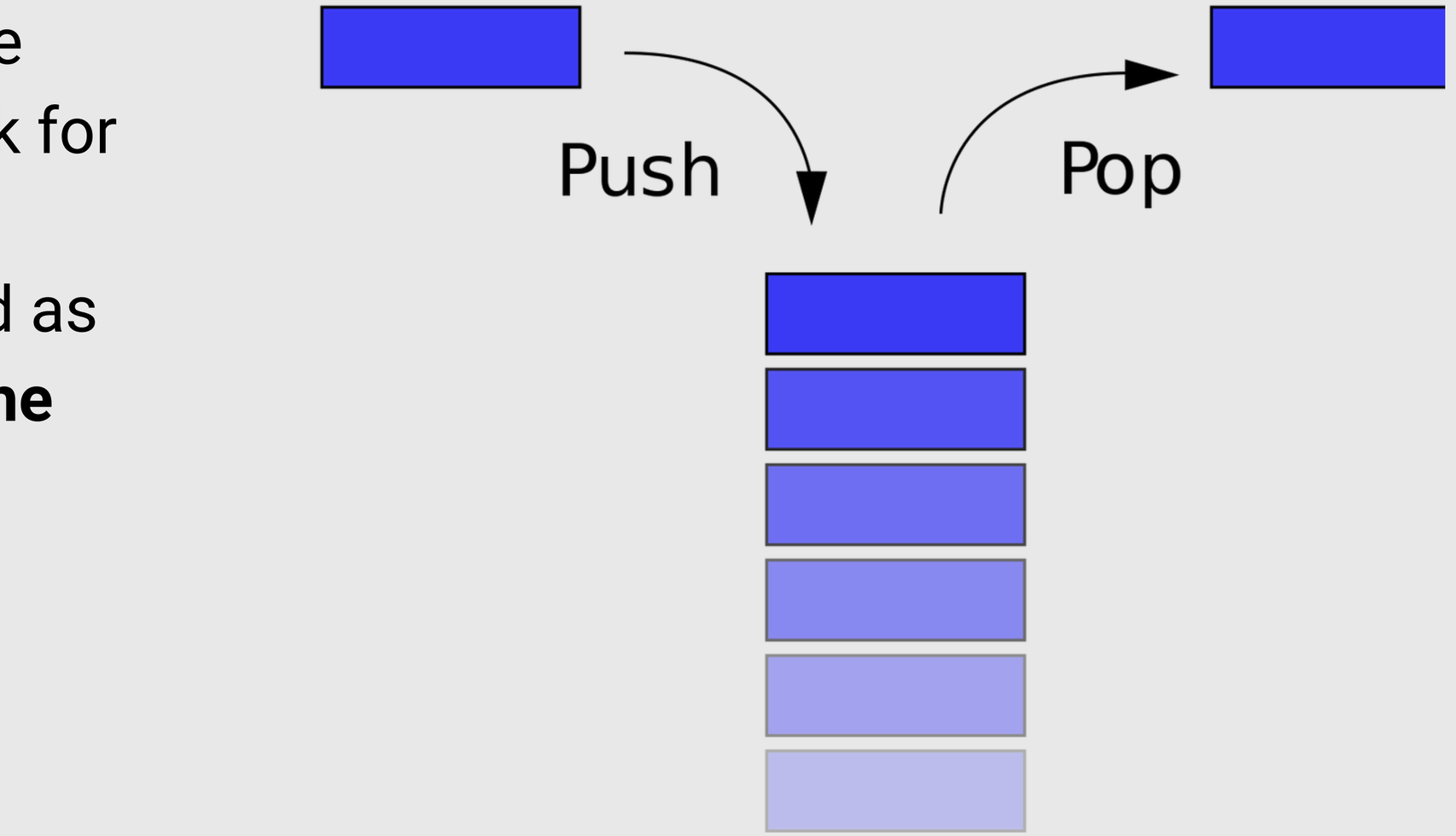
Team Members:

1. Jai Bhadu
2. Nitin Raj
3. Madhav Deorah
4. Hemani Konkati
5. Samrat Middha

WHAT IS A STACK BASED INSTRUCTION SET?

LIFO **STACK** FOR COMPUTATION

Stack based ISAs use Last in First out stack for computation , where **operands** are defined as elements on **top of the stack**



HOW IS STACK BASED ISA DIFFERENT FROM CONVENTIONAL ISA?

01

NO EXPLICIT OPERANT **REGISTERS**

Operands are taken from TOS

02

SIMPLER TO IMPLEMENT

Can be less efficient in some areas

03

SMALLER PROGRAM SIZE

implicit operands insure small size

04

VERSATILE FOR HARDWARE

Easy to implement on a wide variety of hardware

REGISTER ISA VS STACK BASED ISA

```
1   .main
2
3       MOV R1, 1
4       MOV R2, 1
5       ADD R1, R2
6       INC R1
7       MOV R2, 2
8       MUL R1, R2
9       INC R1
```

```
1   .main:
2
3       push 1
4       push 1
5       add
6       inc
7       push 2
8       mul
9       inc
```

SIMPLER AND MORE **COMPACT** INSTRUCTIONS

"Although virtually every processor today uses a loadstore register architecture, stack architectures attract attention again due to the success of Java.."

MARTIN SCHOEBERL

OUR APPROACH TO A STACK BASED INSTRUCTION SET ARCHITECTURE

1: **CUSTOM ISA**

Custom ISA with a variety of instructions including branch instructions implemented in **C++.**

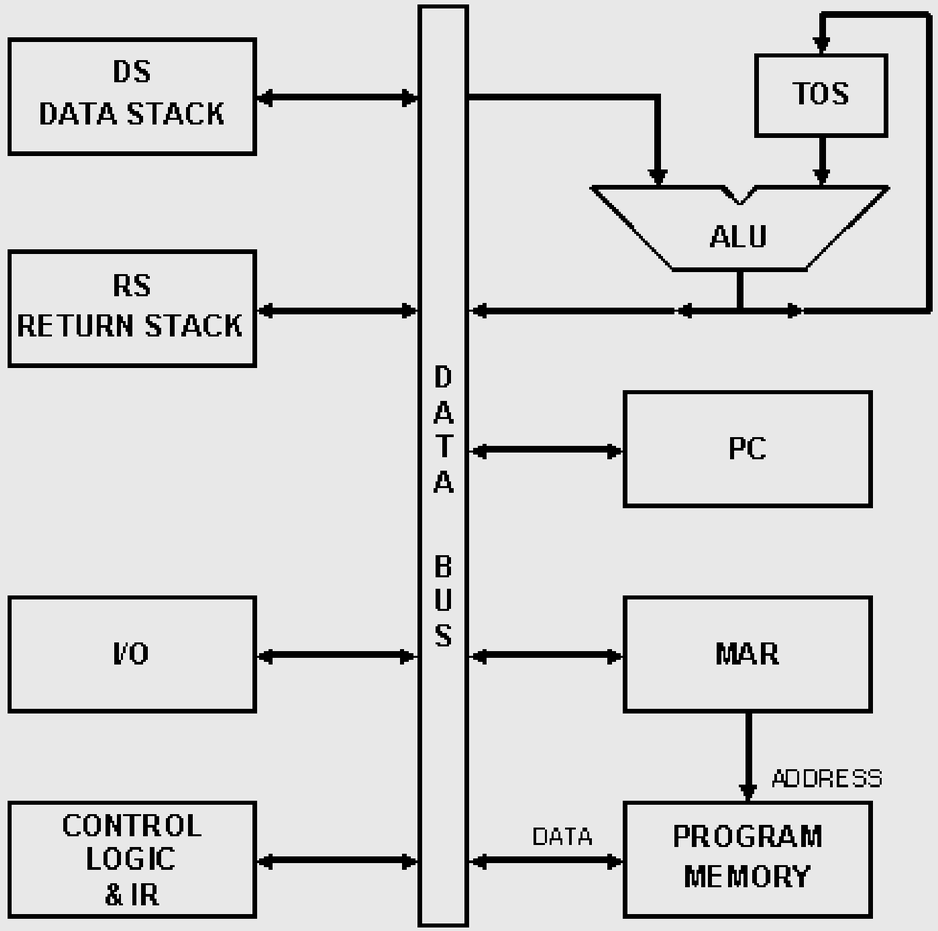
2: **MULTIPLE STACKS**

Contains **DataStack , Return Stack** , ALU, IO/Control module and a Program Memory.

3: **VISUALISATION/DEBUGGER**

Visualisation of Stacks and memory in real time **Using SDL (Simple DirectMedia Layer).**

HOW ARE INSTRUCTIONS EXECUTED



1. Instructions are retrieved from the program memory by the **PC**
2. Now, instead of registers the **ALU** manipulates the **TOS**
3. The operands are fetched from **TOS** and Result is saved in **TOS**
4. The **Return Stack** stores the Return Addresses for call instructions

INSTRUCTION SET

ARITHMETIC AND LOGICAL

1. add
2. sub
3. mul
4. div
5. inc
6. dec
7. mod
8. and
9. or
10. xor

STACK MANIPULATION

1. push <op>
2. pop
3. pushr <op>
4. popr
5. dup
6. swap
7. over
8. drop

CONTROL AND BRANCH

1. cmp
2. b
3. beq
4. bgt
5. call
6. ret
7. halt

LIMITS OF A STACK BASED ISA

- HOW TO USE **LOOPS ?**
- HOW TO USE **RECURSION?**
- HOW TO WRITE **COMPLEX APPLICATIONS?**

FACTORIAL USING RECURSION

```c
int factorial(int n){
    if( n==1 ){
        return 1;
    }
    else return n*factorial(n-1);
}
int main(){
    printf("%d",factorial(5));
}
```

```
.FACTORIAL:
    push 1
    cmp
    beq .base
    pop
    dup
    push 1
    sub
    call .FACTORIAL
    mul
    ret
.base:
    pop
    ret
.main:
    push 5
    call .FACTORIAL
    print
halt
```

DISADVANTAGES

- **Difficult to implement** even simple loops and recursive functions
- Implementations are very **inefficient**
- You Lose **Readability and Efficiency**

ONE SOLUTION:

Use Variables

VARIABLES

1. Var <var>
2. store <var>
3. load <var>

```
.FACTORIAL:
   load n
   push 1
   cmp
   pop
   beq .BASE_CASE

   load n
   push 1
   sub
   store n
   call .FACTORIAL
   mul
   ret

.BASE_CASE:
   ret

.main:
   var n
   push 5
   store n
   call .FACTORIAL
   print
halt
```

- Variables need to be **accessed from memory every time.**
- Still doesn’t address the issue of **efficiency.**
- Use of Variables requires either **complex pattern of dup , push , pop** or explicit store/load instruction

EXAMPLE PROGRAMS

```c
#include <stdio.h>

int sum(int n) {
    if (n <= 0) return 0;
    return n + sum(n - 1);
}

int main() {
    int n = 10;
    printf("%d\n", sum(n));
    return 0;
}
```

```
.SUM_UP_TO_N:
    dup
    push 0
    cmp
    pop
    beq .BASE_CASE_SUM

    dup
    push 1
    sub
    call .SUM_UP_TO_N
    add
    ret

.BASE_CASE_SUM:
    pop
    push 0
    ret

.main:
    push 10
    call .SUM_UP_TO_N
    print
halt
```

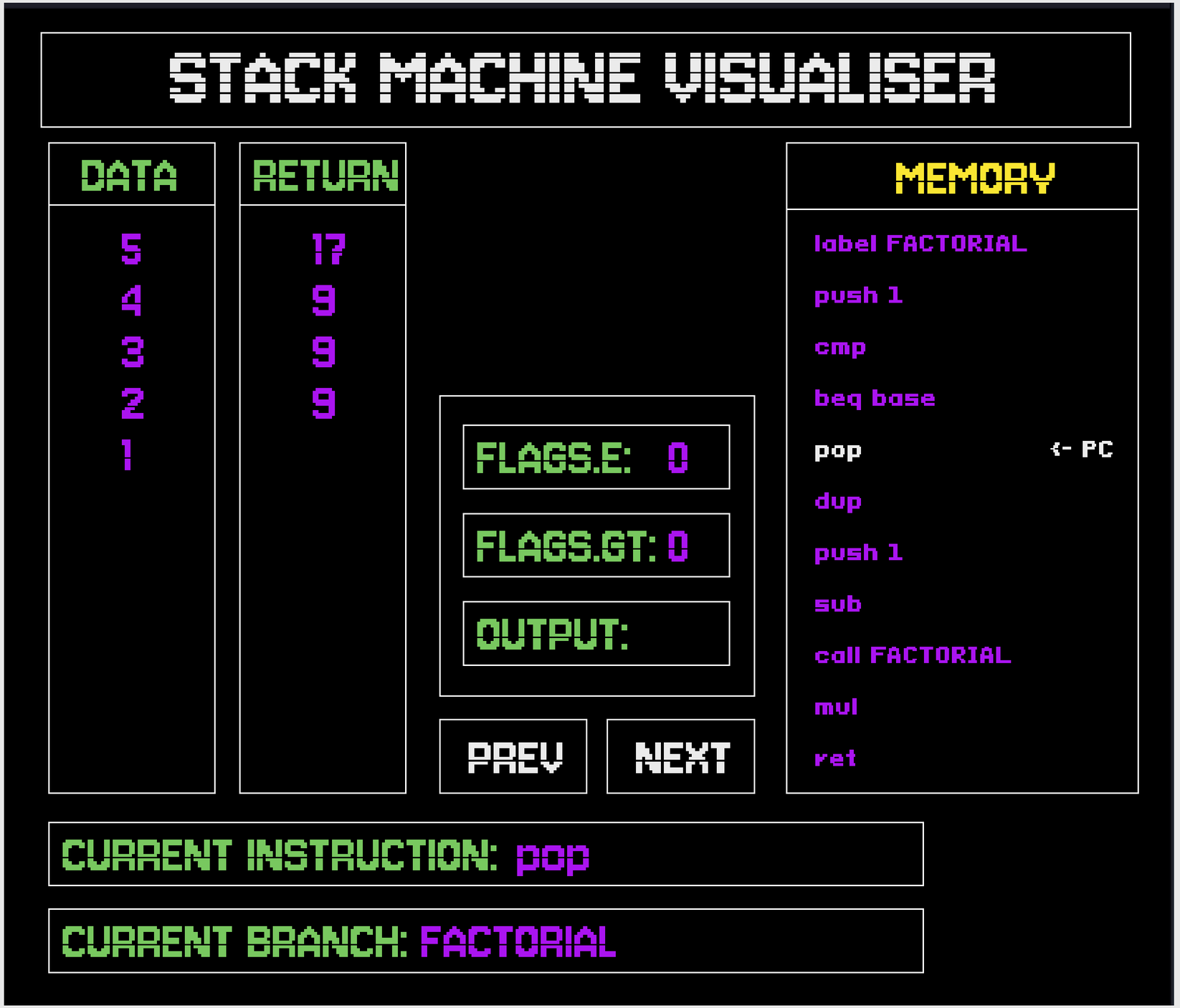
EXAMPLE PROGRAMS

```c
#include <stdio.h>

void printMultiplicationTable(int n) {
    for (int i = 1; i <= 10; ++i) {
        printf("%d x %d = %d\n", n, i, n * i);
    }
}

int main() {
    int n = 5;
    printMultiplicationTable(n);
    return 0;
}
```

```
.MULTIPLICATION_TABLE:
   push 10
   cmp
   pop
   bgt .END_TABLE

   dup
   push 1
   add
   dup
   push 1
   sub
   mul
   print

   call .MULTIPLICATION_TABLE

.END_TABLE:
   ret

.main:
   push 5
   push 1
   call .MULTIPLICATION_TABLE
halt
```

VISUALISATION

- View of **Memory** , **Data Stack** , **Return Stack** and **PC** after each instruction
- View **Execution** and go back or go forward **at each step**
- Implemented in **SDL2**
- Useful for **visualising** and **debugging** code


STACK MACHINE VISUALISER
DATA
5
4
3
2
1
RETURN
17
9
9
9
FLAGS.E: 0
FLAGS.GT: 0
OUTPUT:
PREV
NEXT
MEMORY
label FACTORIAL
push 1
cmp
beq base
pop
<- PC
dup
push 1
sub
call FACTORIAL
mul
ret
CURRENT INSTRUCTION: pop
CURRENT BRANCH: FACTORIAL