CS322 MINI PROJECT
EXTENDING FUNCTIONALITY OF MARS MIPS ASSEMBLER

REPORT

STACK MEMORY:

Stack is one of the Memory segments in the main memory layout. The role of stack is the storage of temporary data while handling function calls, storage for local variables, passing of parameters in function calls, saving of registers during exception sequences etc.

Stack is mainly used in case of nested subroutine calls, where each of the subroutine may have a set of variables which are local to that subroutine, i.e. their scope is limited only to that function/subroutine call. This temporary data can be conveniently stored on the stack in a stack frame.

STACK FRAME:

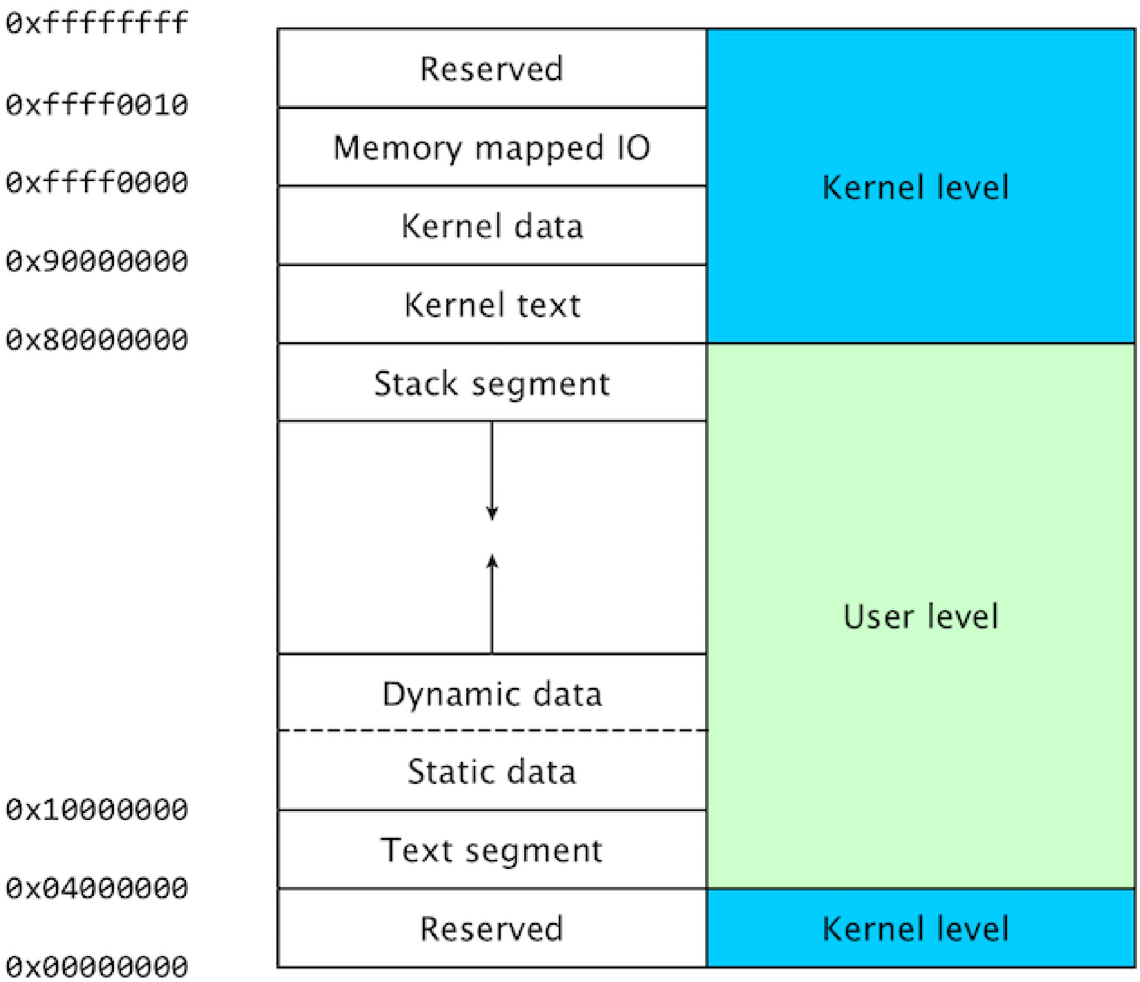
Each function call gets its memory allocated on the stack in the form of a stack frame. Once the function has completed its execution, its stack frame(which contains all the pass arguments and local variables) gets erased from the main memory.

When a function is called, it creates a new frame onto the stack, which will be used for local storage. Before the function returns, it must pop its stack frame, to restore the stack to its original state.

THE MIPS STACK:

In MIPS machines, part of main memory is reserved for a stack. The stack grows downward in terms of memory addresses. — The address of the top element of the stack is stored (by convention) in the "stack pointer" register, $sp.

MIPS does not provide “push” and “pop” instructions. Instead, they must be done explicitly by the programmer.

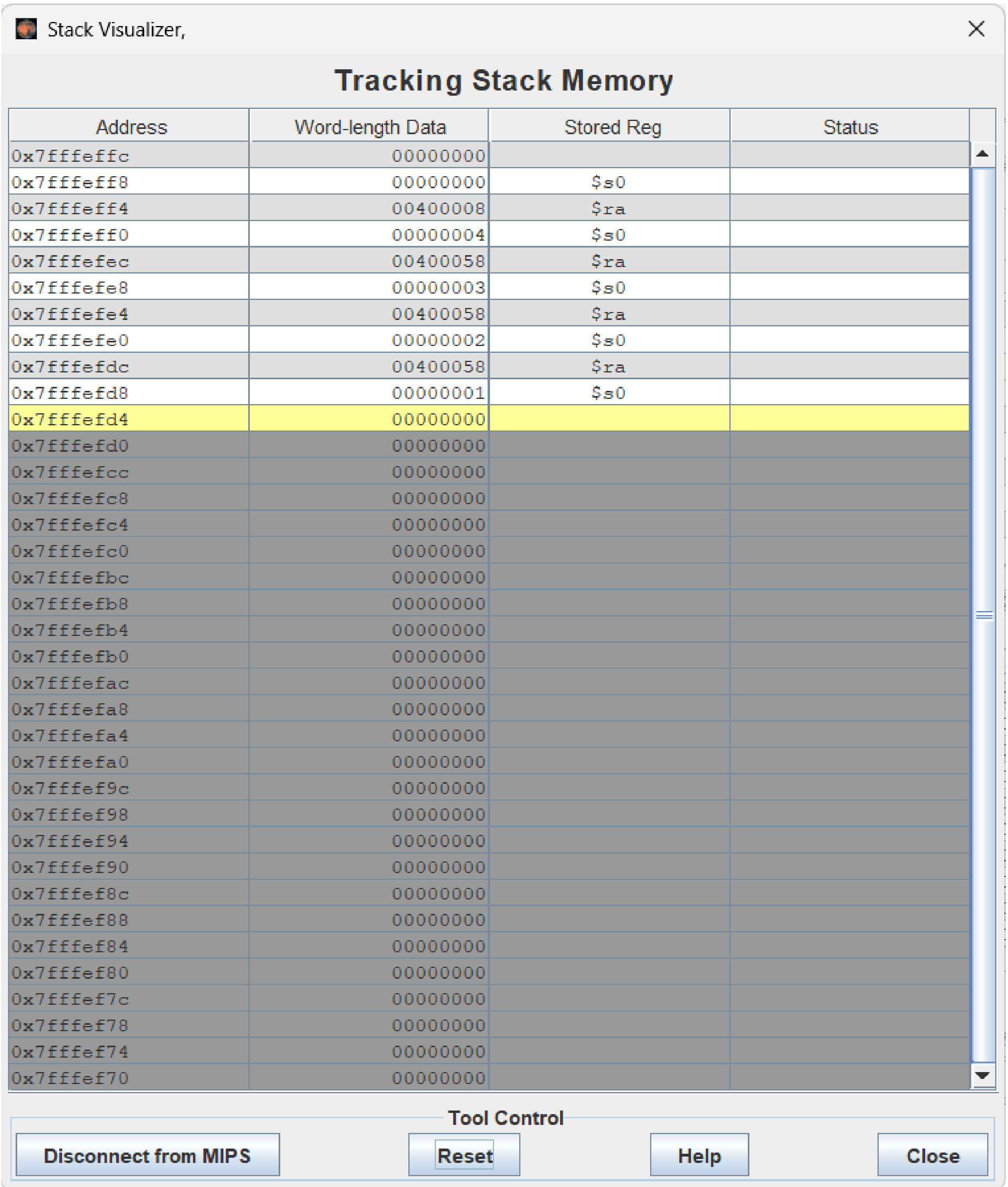


MIPS memory layout

<u>MARS STACK VISUALIZER TOOL:</u>

In this project, I have implemented an in-built tool by extending the Abstract Tools class in MARS, which can visually simulate the stack memory in between function calls. It keeps track of:

- Memory address referenced by the stack pointer
- Registered being saved in the stack
- Value or address contained in the given register



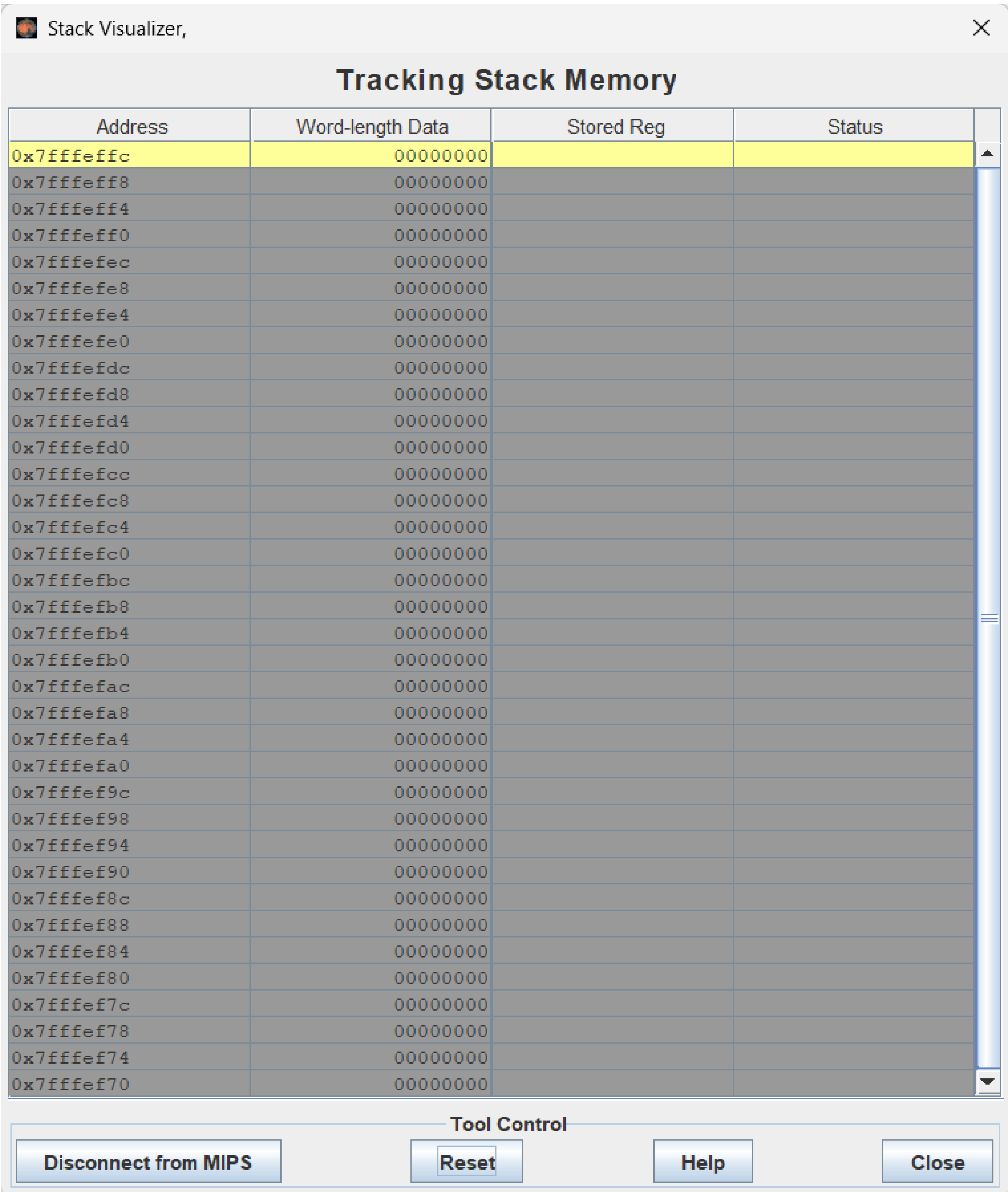
Address	Word-length Data	Stored Reg	Status
0x7fffeffc	00000000		
0x7fffeff8	00000000	$s0	
0x7fffeff4	00400008	$ra	
0x7fffeff0	00000004	$s0	
0x7fffefec	00400058	$ra	
0x7fffefe8	00000003	$s0	
0x7fffefe4	00400058	$ra	
0x7fffefe0	00000002	$s0	
0x7fffefdc	00400058	$ra	
0x7fffefd8	00000001	$s0	
0x7fffefd4	00000000		
0x7fffefd0	00000000		
0x7fffefcc	00000000		
0x7fffefc8	00000000		
0x7fffefc4	00000000		
0x7fffefc0	00000000		
0x7fffefbc	00000000		
0x7fffefb8	00000000		
0x7fffefb4	00000000		
0x7fffefb0	00000000		
0x7fffefac	00000000		
0x7fffefa8	00000000		
0x7fffefa4	00000000		
0x7fffefa0	00000000		
0x7fffef9c	00000000		
0x7fffef98	00000000		
0x7fffef94	00000000		
0x7fffef90	00000000		
0x7fffef8c	00000000		
0x7fffef88	00000000		
0x7fffef84	00000000		
0x7fffef80	00000000		
0x7fffef7c	00000000		
0x7fffef78	00000000		
0x7fffef74	00000000		
0x7fffef70	00000000		

The highlighted (yellow) row indicates the current stack pointer location and the greyed rows have not been allocated yet. The rows above the $sp show the saved register name, value and the address in stack memory.

SIMULATION AND ANALYSIS:

For demonstration purpose, I will simulate the tool using the MIPS codes for:

1. Calculating factorial sequence
2. Calculating fibonacci sequence
3. Calculating fibonacci sequence using Dynamic Programming



Stack Visualizer,

Tracking Stack Memory

Address	Word-length Data	Stored Reg	Status
0x7fffeffc	00000000		
0x7fffeff8	00000000		
0x7fffeff4	00000000		
0x7fffeff0	00000000		
0x7fffefec	00000000		
0x7fffefe8	00000000		
0x7fffefe4	00000000		
0x7fffefe0	00000000		
0x7fffefdc	00000000		
0x7fffefd8	00000000		
0x7fffefd4	00000000		
0x7fffefd0	00000000		
0x7fffefcc	00000000		
0x7fffefc8	00000000		
0x7fffefc4	00000000		
0x7fffefc0	00000000		
0x7fffefbc	00000000		
0x7fffefb8	00000000		
0x7fffefb4	00000000		
0x7fffefb0	00000000		
0x7fffefac	00000000		
0x7fffefa8	00000000		
0x7fffefa4	00000000		
0x7fffefa0	00000000		
0x7fffef9c	00000000		
0x7fffef98	00000000		
0x7fffef94	00000000		
0x7fffef90	00000000		
0x7fffef8c	00000000		
0x7fffef88	00000000		
0x7fffef84	00000000		
0x7fffef80	00000000		
0x7fffef7c	00000000		
0x7fffef78	00000000		
0x7fffef74	00000000		
0x7fffef70	00000000		

Initial state of the visualizer

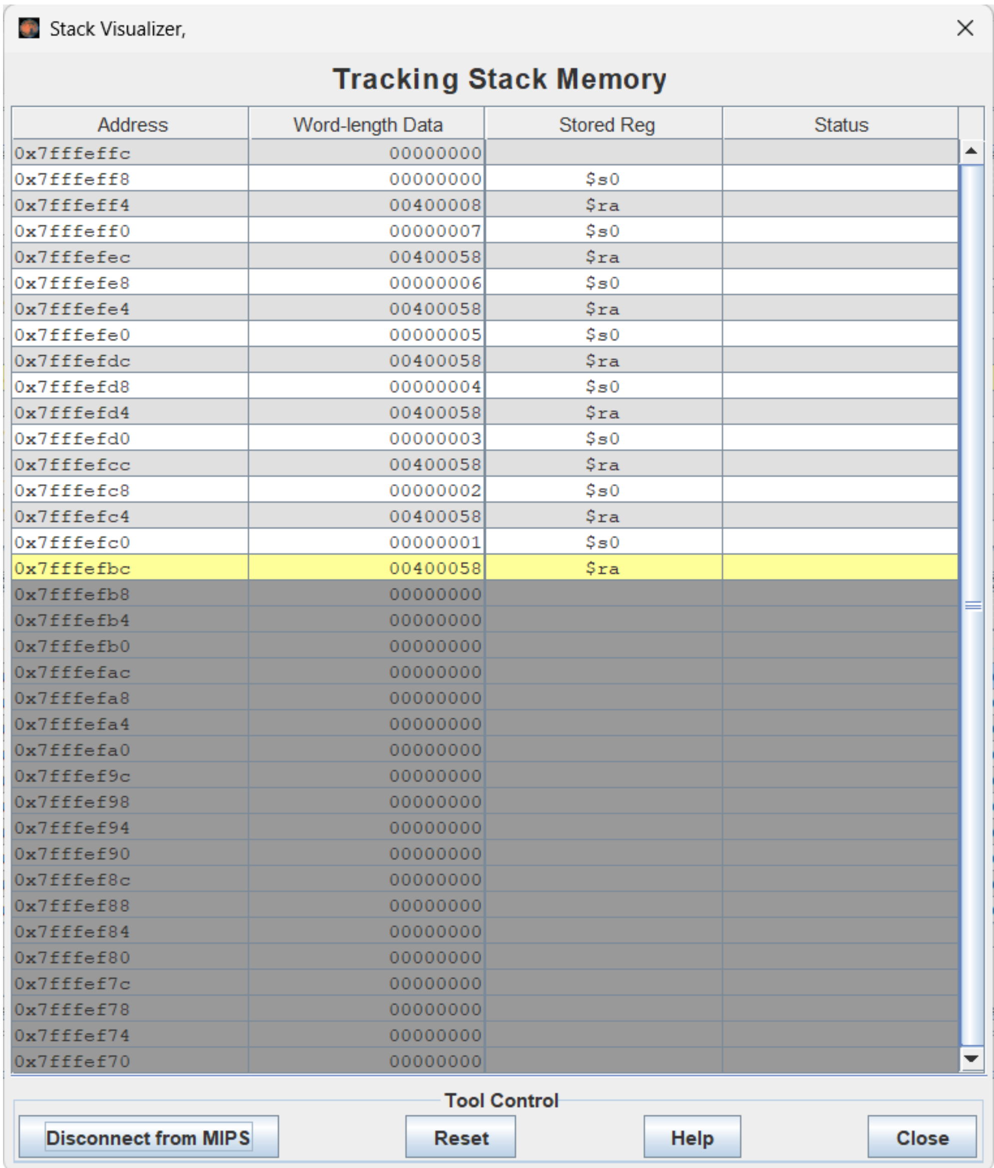
FACTORIAL RECURSIVE CODE:

The process of calculating factorial(N) consists of recursively calling the function alongside reducing the value of N as follows:

$$\text{factorial}(N) = N * \text{factorial}(N-1)$$

where the base case is N = 0, which returns the value 1.

Thus, the value of N gets stored on the stack as a function parameter along with the return value in a stack frame. When the stack reaches its max depth, we see that the stack starts unrolling and consequently the frames start getting deleted in a sequence.



Address	Word-length Data	Stored Reg	Status
0x7fffeffc	00000000		
0x7fffeff8	00000000	$s0	
0x7fffeff4	00400008	$ra	
0x7fffeff0	00000007	$s0	
0x7fffefec	00400058	$ra	
0x7fffefe8	00000006	$s0	
0x7fffefe4	00400058	$ra	
0x7fffefe0	00000005	$s0	
0x7fffefdc	00400058	$ra	
0x7fffefd8	00000004	$s0	
0x7fffefd4	00400058	$ra	
0x7fffefd0	00000003	$s0	
0x7fffefcc	00400058	$ra	
0x7fffefc8	00000002	$s0	
0x7fffefc4	00400058	$ra	
0x7fffefc0	00000001	$s0	
0x7fffefbc	00400058	$ra	
0x7fffefb8	00000000		
0x7fffefb4	00000000		
0x7fffefb0	00000000		
0x7fffefac	00000000		
0x7fffefa8	00000000		
0x7fffefa4	00000000		
0x7fffefa0	00000000		
0x7fffef9c	00000000		
0x7fffef98	00000000		
0x7fffef94	00000000		
0x7fffef90	00000000		
0x7fffef8c	00000000		
0x7fffef88	00000000		
0x7fffef84	00000000		
0x7fffef80	00000000		
0x7fffef7c	00000000		
0x7fffef78	00000000		
0x7fffef74	00000000		
0x7fffef70	00000000		

At max stack depth (for N = 7), we observe 7 stack frames

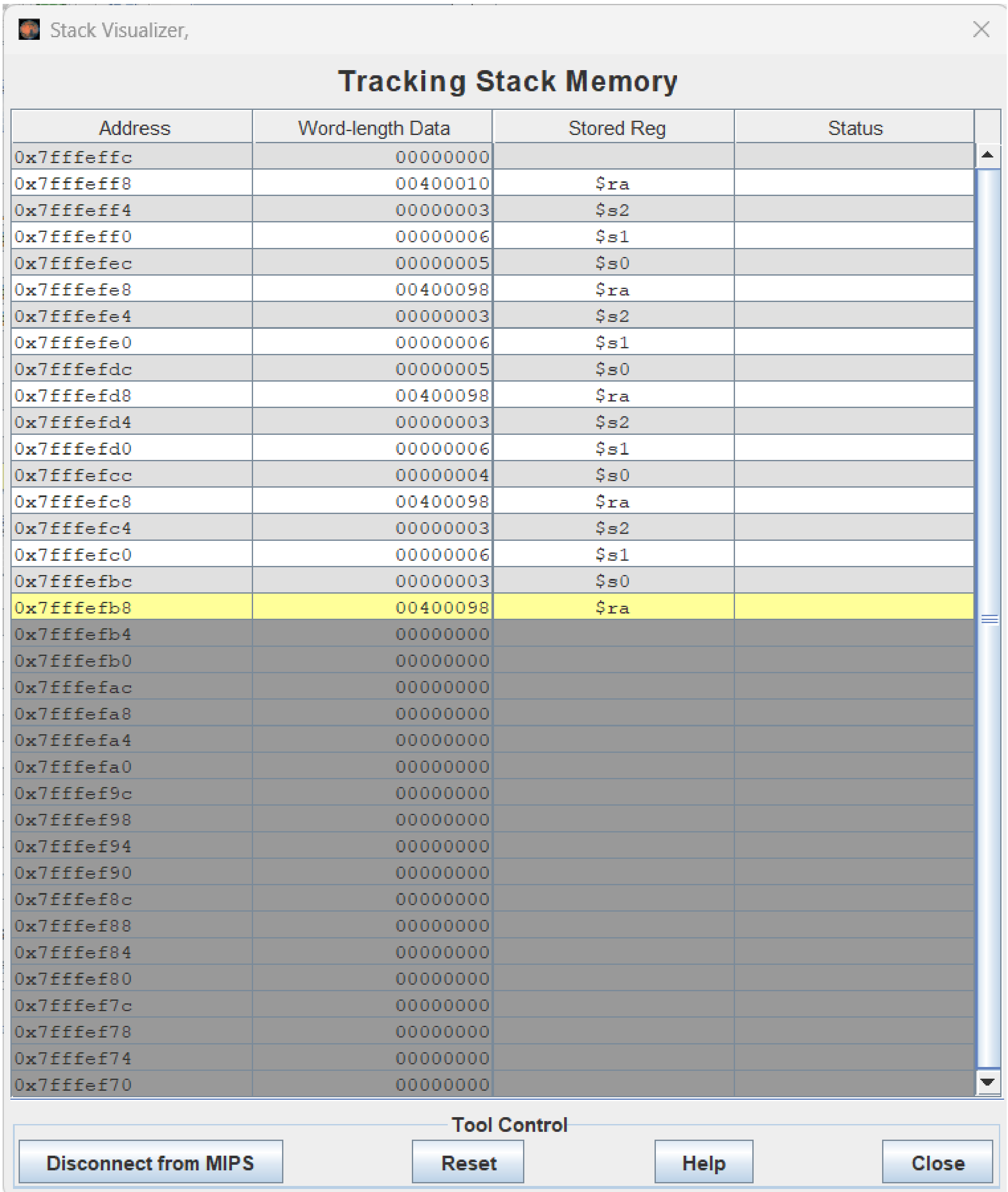
FIBONACCI RECURSIVE CODE:

We recursively calling the function as follows:

$$fib(N) = fib(N-1) + fib(N-2)$$

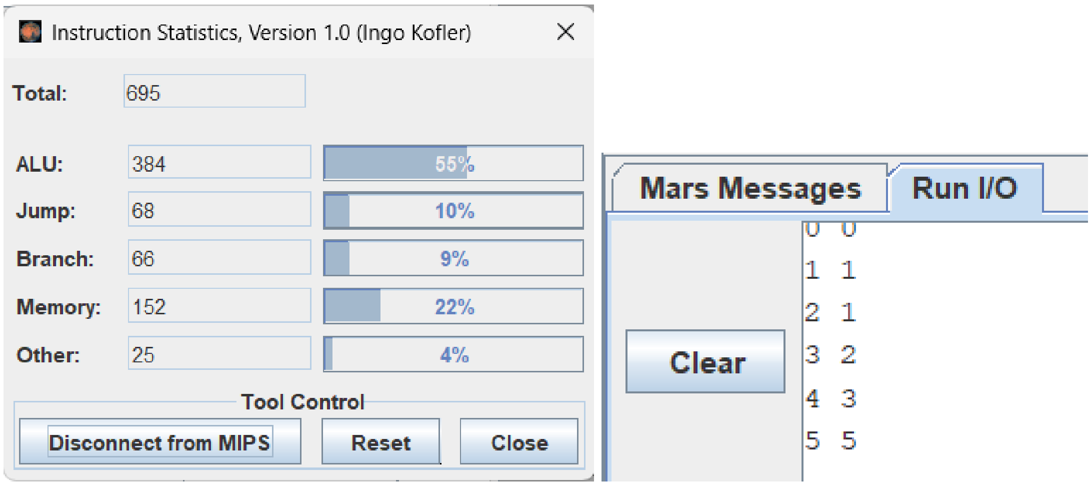
where the base case is: fib(0) = 0 and fib(1) = 1

Thus, we store the value belonging to one function call in one frame stack which gets deleted after the unwinding of stack starts.



Address	Word-length Data	Stored Reg	Status
0x7fffeffc	00000000		
0x7fffeff8	00400010	$ra	
0x7fffeff4	00000003	$s2	
0x7fffeff0	00000006	$s1	
0x7fffefec	00000005	$s0	
0x7fffefe8	00400098	$ra	
0x7fffefe4	00000003	$s2	
0x7fffefe0	00000006	$s1	
0x7fffefdc	00000005	$s0	
0x7fffefd8	00400098	$ra	
0x7fffefd4	00000003	$s2	
0x7fffefd0	00000006	$s1	
0x7fffefcc	00000004	$s0	
0x7fffefc8	00400098	$ra	
0x7fffefc4	00000003	$s2	
0x7fffefc0	00000006	$s1	
0x7fffefbc	00000003	$s0	
0x7fffefb8	00400098	$ra	
0x7fffefb4	00000000		
0x7fffefb0	00000000		
0x7fffefac	00000000		
0x7fffefa8	00000000		
0x7fffefa4	00000000		
0x7fffefa0	00000000		
0x7fffef9c	00000000		
0x7fffef98	00000000		
0x7fffef94	00000000		
0x7fffef90	00000000		
0x7fffef8c	00000000		
0x7fffef88	00000000		
0x7fffef84	00000000		
0x7fffef80	00000000		
0x7fffef7c	00000000		
0x7fffef78	00000000		
0x7fffef74	00000000		
0x7fffef70	00000000		

At max depth (for N = 5)

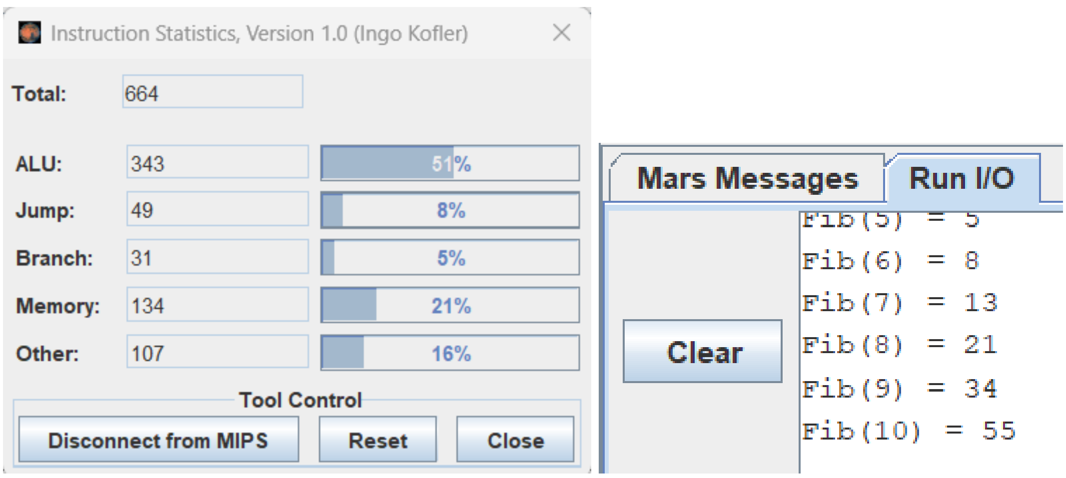


(N = 5)

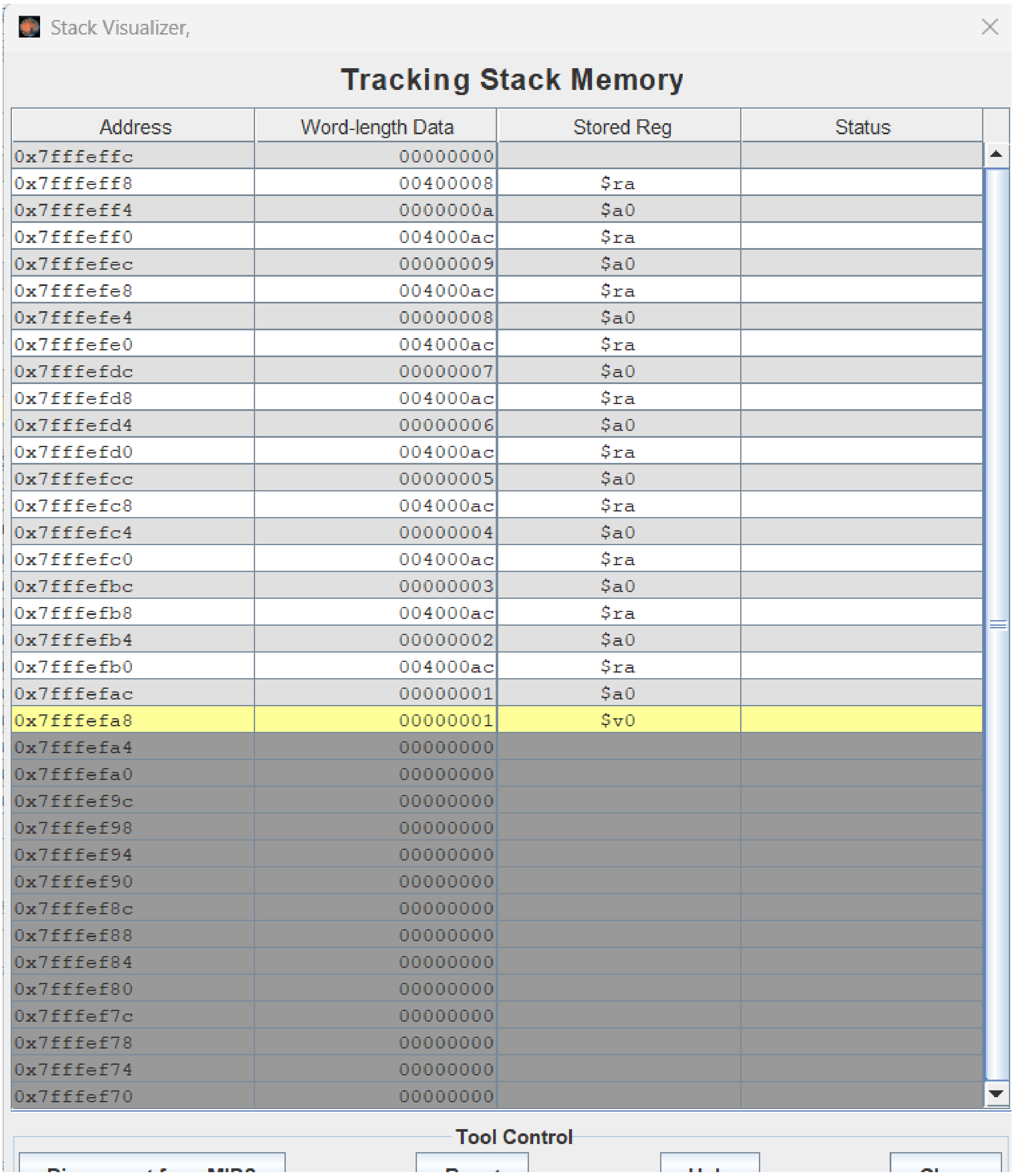
<u>FIBONACCI RECURSIVE CODE WITH DYNAMIC PROGRAMMING:</u>

The procedure remains the same as in general fibonacci sequence calculation, except we first check if a particular value has been calculated and stored in the data memory before. We call (and thus allocate a stack frame) only if the function has not been called with that argument yet.

This makes our program much more efficient in terms of time complexity as well as the stack memory used.



(N = 10)



Address	Word-length Data	Stored Reg	Status
0x7fffeffc	00000000		
0x7fffeff8	00400008	$ra	
0x7fffeff4	0000000a	$a0	
0x7fffeff0	004000ac	$ra	
0x7fffefec	00000009	$a0	
0x7fffefe8	004000ac	$ra	
0x7fffefe4	00000008	$a0	
0x7fffefe0	004000ac	$ra	
0x7fffefdc	00000007	$a0	
0x7fffefd8	004000ac	$ra	
0x7fffefd4	00000006	$a0	
0x7fffefd0	004000ac	$ra	
0x7fffefcc	00000005	$a0	
0x7fffefc8	004000ac	$ra	
0x7fffefc4	00000004	$a0	
0x7fffefc0	004000ac	$ra	
0x7fffefbc	00000003	$a0	
0x7fffefb8	004000ac	$ra	
0x7fffefb4	00000002	$a0	
0x7fffefb0	004000ac	$ra	
0x7fffefac	00000001	$a0	
0x7fffefa8	00000001	$v0	
0x7fffefa4	00000000		
0x7fffefa0	00000000		
0x7fffef9c	00000000		
0x7fffef98	00000000		
0x7fffef94	00000000		
0x7fffef90	00000000		
0x7fffef8c	00000000		
0x7fffef88	00000000		
0x7fffef84	00000000		
0x7fffef80	00000000		
0x7fffef7c	00000000		
0x7fffef78	00000000		
0x7fffef74	00000000		
0x7fffef70	00000000		

At max depth (For N = 10)

It can be noted that dynamic programming makes the program much more efficient since it does not need to call the same routine multiple times redundantly if the return value for that has been stored in a data memory location retrieved from the stack before the frame can be erased.

SUMMARY:

I have created a Stack Visualizer tool which keeps track of the stack pointer and visually simulates the stack memory in between nested function calls, storing local variables, caller and callee-saved registers, function parameters and arguments on the stack, in the form of stack frames which are erased at the end of function execution.

SUBMITTED BY: Animesh Kumar Sinha
ROLL NUMBER: 2001CS07