AMRITA VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

19AIE113
ELEMENTS OF COMPUTING SYSTEM -2

Memory Allocation Strategies in Jack

PRESENTED BY :-

Anirudh Edpuganti	*- CB.EN.U4AIE20005*
Bhoomika M	*- CB.EN.U4AIE20008*
Gokul R	*- CB.EN.U4AIE20018*
Rishekesan SV	*- CB.EN.U4AIE20058*
Shreya Sanghamitra	*- CB.EN.U4AIE20066*

Acknowledgement

We would like to express our special thanks of gratitude to our teacher (Dr Jyothish Sir) who gave us the golden opportunity to do this wonderful project on the topic (Implementation of memory allocations in Jack), which also helped us in doing a lot of research and we came to know about so many new things. We are thankful for the opportunity given.

Declaration

We declare that the Submitted Report is our original work and no values and context of it have been copy-pasted from anywhere else. We take full responsibility, that if in future, the report is found invalid or copied, the last decision will be of the Faculty concerned. Any form of plagiarism will lead to the disqualification of the report.

Table of Contents

Aim

Implement the storage allocation strategies in Jack
1. Heap
2. Stack
3. Static

Introduction

One of the most basic problems that every compiler faces is figuring out how to translate the various types of variables in the source programme to the target platform's memory. This isn't a simple job. For starters, different types of variables demand different amounts of memory, thus the mapping is not one-to-one. Second, various types of variables have varying life periods. A single duplicate of each, for example. For the length of the programme, a static variable should be kept "alive." Each object instance of the class, on the other hand, should contain a unique copy of all its instance variables (fields).

In addition, each time a function is called, a new copy of the function's local variables must be produced — a requirement that is evident in recursion.

Heap

The heap is a section of memory reserved for dynamic allocation. The allocation and deallocation of blocks from the heap, unlike the stack, has no prescribed pattern; you can allocate and free a block at any moment. This makes keeping track of which parts of the heap are allocated or released at any one time much more difficult; there are a variety of custom heap allocators available to tweak performance for different usage patterns.

Stack

The stack is a global data structure that is used to save and restore all of the VM functions' resources (memory segments) as they move up the calling hierarchy (e.g. main and factorial). The working stack (also known as the "stack frame") of the current function is at the top of this stack. All operations are performed on the stack.

Static

When a new symbol is encountered for the first time in an assembly program, the assembler allocates a new RAM address to it, starting at address 16.
This stores the static variables shared by all the functions in the same .vm file and only has one copy of each static variable and this copy is shared by all the object instances of the class. This single copy of each static variable should be kept alive during the complete duration of the program’s run-time.

Software Suite

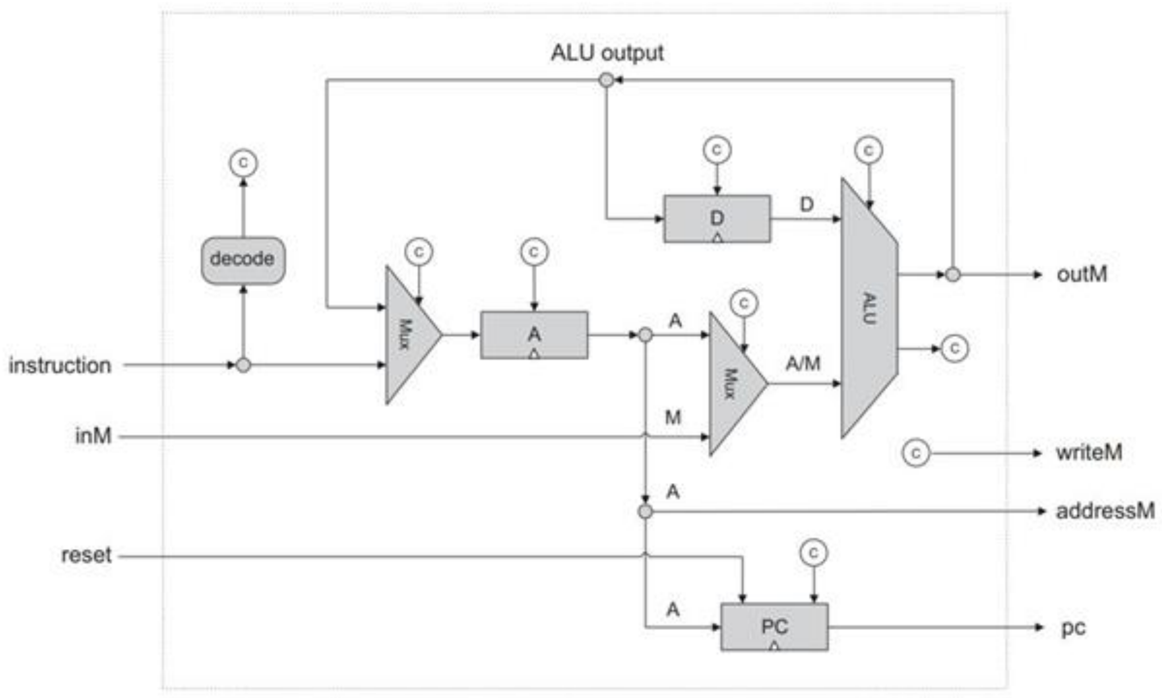
- Nand2Tetris

Theory

CPU (Central Processing Unit)

A central processing unit is an electronic circuitry within a computer that executes instructions that make up a computer program.

The control unit controls all CPU operations, including ALU operations, the movement of data within the CPU, and the exchange of data and control signals across external interfaces.



As we know the CPU will only read the instruction memory and executes the instruction, and will not write or modify instruction to the memory.

There are three parts connected to the CPU -instruction memory, data memory and user.

Instruction Memory - It contains a 16 bit 'instruction' as input and the output of PC can be explained based on Instruction memory.

Data Memory - It contains 16-bit input 'intM' and three outputs out, write M, and address.

User - Here the user must give only one input that is 'reset'. If the reset pin is triggered it resets the program counter, the D register and A register. The CPU will take the instruction one by one from the instruction memory and will take the instruction pointed out by the program counter.

So, there will be a connection from the CPU to the instruction memory which is through the program counter. Once the program counter is set to the instruction memory it will send the corresponding instruction then after the CPU has interpreted that instruction it will execute that instruction. As a part of that execution, it can read from the data memory, and it basically collects the data and it can also write into the data memory for that it has to give the address it has to give what data to write and it also have to enable the load of the memory to see that in this particular address the corresponding data is written.

INSTRUCTION BITS

These instruction bits are two types: A instruction and C instructions. These instructions are told apart using the first bit of the instruction which is called the T-pin. If the first bit is "0" then it’s an A instruction and if the first bit is "1" it’s a C instruction.

In the other 15 bits of the instructions the A instruction stores the value and the C instruction stores instruction for the operation.

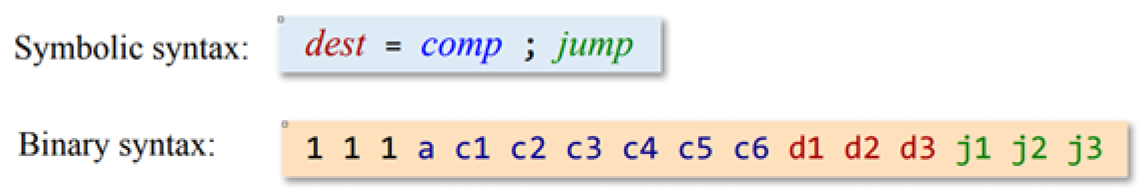
A INSTRUCTION

In A instruction the 15 bits other than the first-bit store a value which is then passed to the A register. This A-register when called gives the value as the output.

C INSTRUCTION

In C instruction the 15 bits are again supposed to be broken down into ALU control bits, Destination bits and jump bits.

CONTROL BITS OR COMPUTATIONAL BITS



Symbolic syntax: *dest* = *comp* ; *jump*

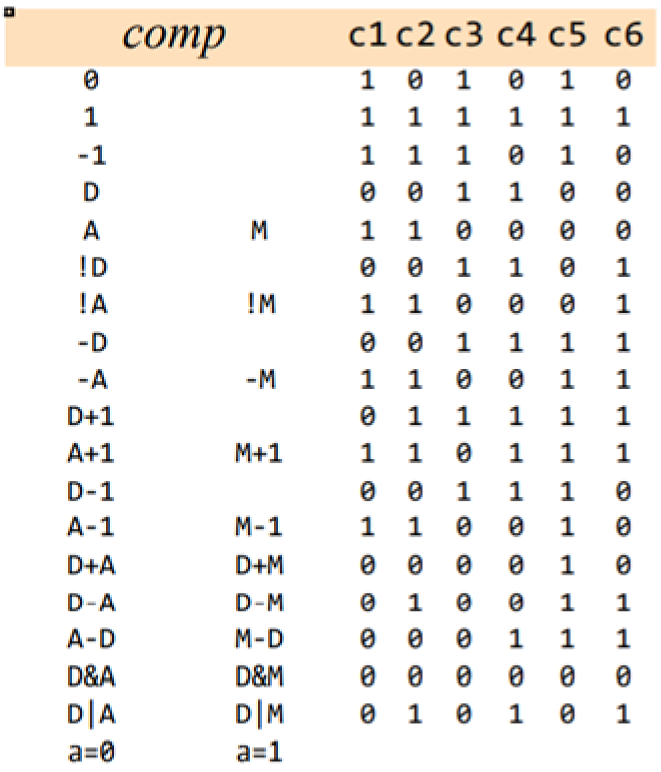
Binary syntax: 1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3

The first three bits are kept constant i.e. "1 1 1".

And the next bit is the "a" bit which is instructed from where the operands are to be taken from.

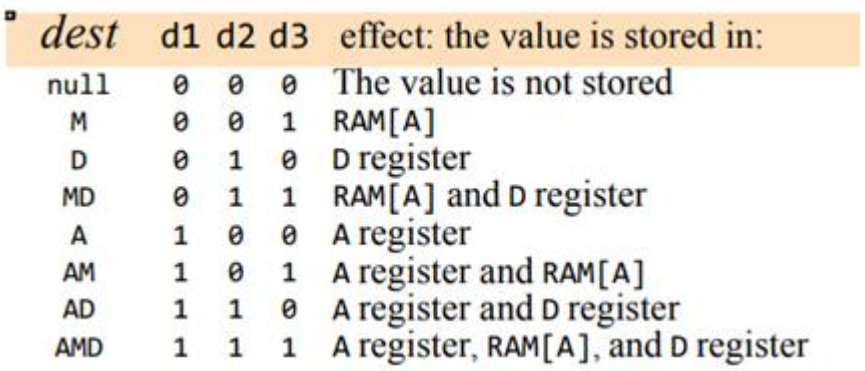
We know that the ALU has two source operands, one operand is always the D register and the other one can either be the A register or the Memory. Here the "a" bit is the op-code is used to refer from where the source operand are to be accessed, i.e. either from A register or from the Memory

The next 6 bits are the computation bits, which is used to compute the operations 16-bit by the user.



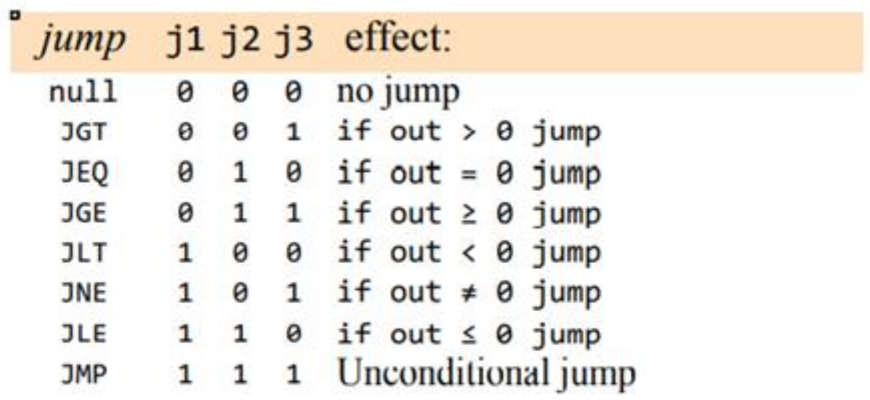
comp		c1	c2	c3	c4	c5	c6
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
a=0	a=1						

The next three bits are the destination bits, these bits control or tell where the result values of the computational part are supposed to get stored.



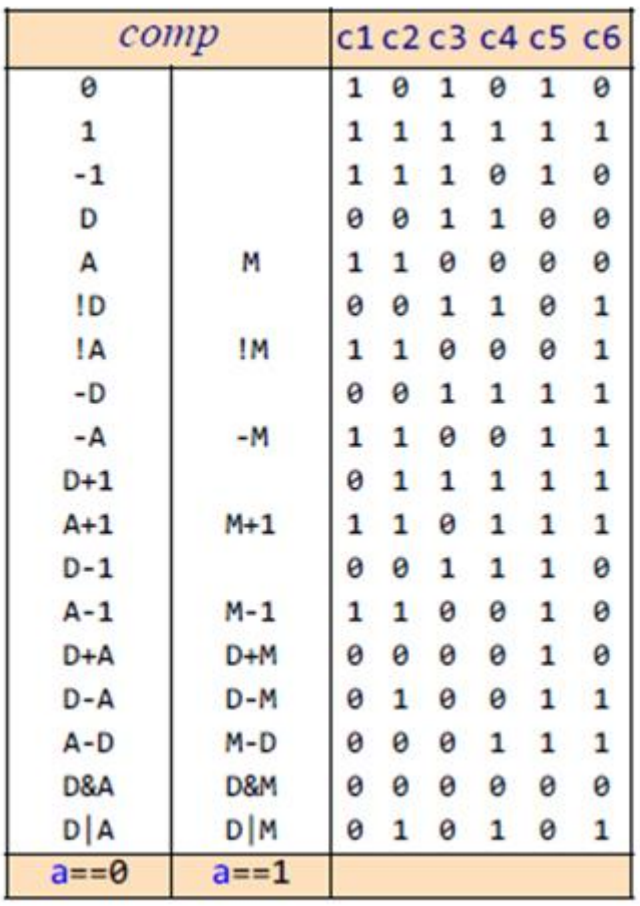
dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

The Last Three bits are the Jump statement bits, this bit instructs the pointer to go to specific memory instruction if the given condition satisfies.



jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Lets us see how these CPU works



comp		c1	c2	c3	c4	c5	c6
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
a==0	a==1						

<u>ALU</u>

The 5th to 10th bit is given to the ALU from the register as an input which it requires to compute.

The ALU has two operands X and Y on which it performs the operations. We may consider X as the value from the D register and Y either from the A register or from the memory depending on the a-bit in the C instruction. So here we use mux16 to decide which it is going to access.

The output of the ALU goes to the memory and is feed to the D register and the A register via 2 Muxes

PC Counter

The program counter does three major operations, Reset, Jump and Increment.

In Reset, the program counter is set to become zero. This reset has the highest priority and is used to start and restart the execution of the program.

If the three jump bits are 0, then there is no jump and here PC is incremented by one step.

If the current instruction has all the jump bits as 1 then we have an unconditional goto.

In this case, PC = A and it means the PC will emit the address of the next instruction that has to be executed.

If only some of the j bits are one then we have a conditional goto.

Hereby understanding the ALU outputs, either PC will increment or PC = A or it will get to zero or it will get reset.

If a jump comes PC will be loaded and if it is not Reset and it is not a load and if it is a

normal instruction then it is getting incremented. So that increment is always not of load.

CPU Model

```
CHIP CPU {

    IN  inM[16],         // M value input  (M = contents
of RAM[A])
        instruction[16], // Instruction for execution
        reset;           // Signals whether to re-start
the current
                         // program (reset==1) or
continue executing
                         // the current program
```

```
(reset==0).

    OUT outM[16],        // M value output
        writeM,          // Write to M?
        addressM[15],    // Address in data memory (of M)
        pc[15];          // address of next instruction

    PARTS:
     NandNot(in=instruction[15], out=notOp);
    myMux16(a=aluOut, b=instruction, sel=notOp,
out=entryMuxOut);

    NandOr(a=notOp, b=instruction[5], out=intoA);
    ARegister(in=entryMuxOut, load=intoA, out=A,
out[0..14]=addressM);


    NandAnd(a=instruction[15], b=instruction[12],
out=AMSwitch);
    myMux16(a=A, b=inM, sel=AMSwitch, out=AM);

    NandAnd(a=instruction[15], b=instruction[4],
out=intoD);
    DRegister(in=aluOut,load=intoD, out=D);

    myALU(x=D, y=AM, zx=instruction[11],
        nx=instruction[10],
        zy=instruction[9],
        ny=instruction[8],
        f=instruction[7],
        no=instruction[6],
        zr=zrOut,
        ng=ngOut,
        out=aluOut,
        out=outM);
    NandAnd(a=instruction[15], b=instruction[3],
out=writeM);


    NandNot(in=ngOut, out=pos);
    NandNot(in=zrOut, out=nzr);
    NandAnd(a=instruction[15], b=instruction[0],
out=jgt);
    NandAnd(a=pos, b=nzr, out=posnzr);
    NandAnd(a=jgt, b=posnzr, out=ld1);

    NandAnd(a=instruction[15], b=instruction[1],
out=jeq);
    NandAnd(a=jeq, b=zrOut, out=ld2);

    NandAnd(a=instruction[15], b=instruction[2],
out=jlt);
    NandAnd(a=jlt, b=ngOut, out=ld3);

    NandOr(a=ld1, b=ld2, out=ldt);
    NandOr(a=ld3, b=ldt, out=ld);

    myPC(in=A, load=ld, inc=true, reset=reset,
out[0..14]=pc);


}
```

Procedure

Memory allocation and access:

- The static variables of a Jack class are allocated to, and accessed via, the VM’s static segment of the corresponding .vm file.

- The local variables of a Jack subroutine are allocated to, and accessed via, the VM’s local segment.

- Before calling a VM function, the caller must push the function’s arguments onto the stack. If the VM function corresponds to a Jack method, the first pushed argument must be the object on which the method is supposed to operate.

- Within a VM function, arguments are accessed via the VM’s argument segment.

- Access to the fields of this object is acquired within VM functions corresponding to Jack methods or constructors by first directing the VM's this segment to the current object (using "pointer 0") and then accessing specific fields using "this index" references.

- The base of this segment is supplied as the 0th argument to VM functions corresponding to Jack methods, and code to set it is automatically inserted by the compiler at the start of the VM function.

- For constructors, the base of this segment is obtained and set when the space for the object is allocated. The code for this allocation is automatically inserted by the compiler at the beginning of the constructor’s code.

- Within a VM function, access to array entries is obtained by pointing the VM’s that segment to the address of the desired array location

Conditions for implementation

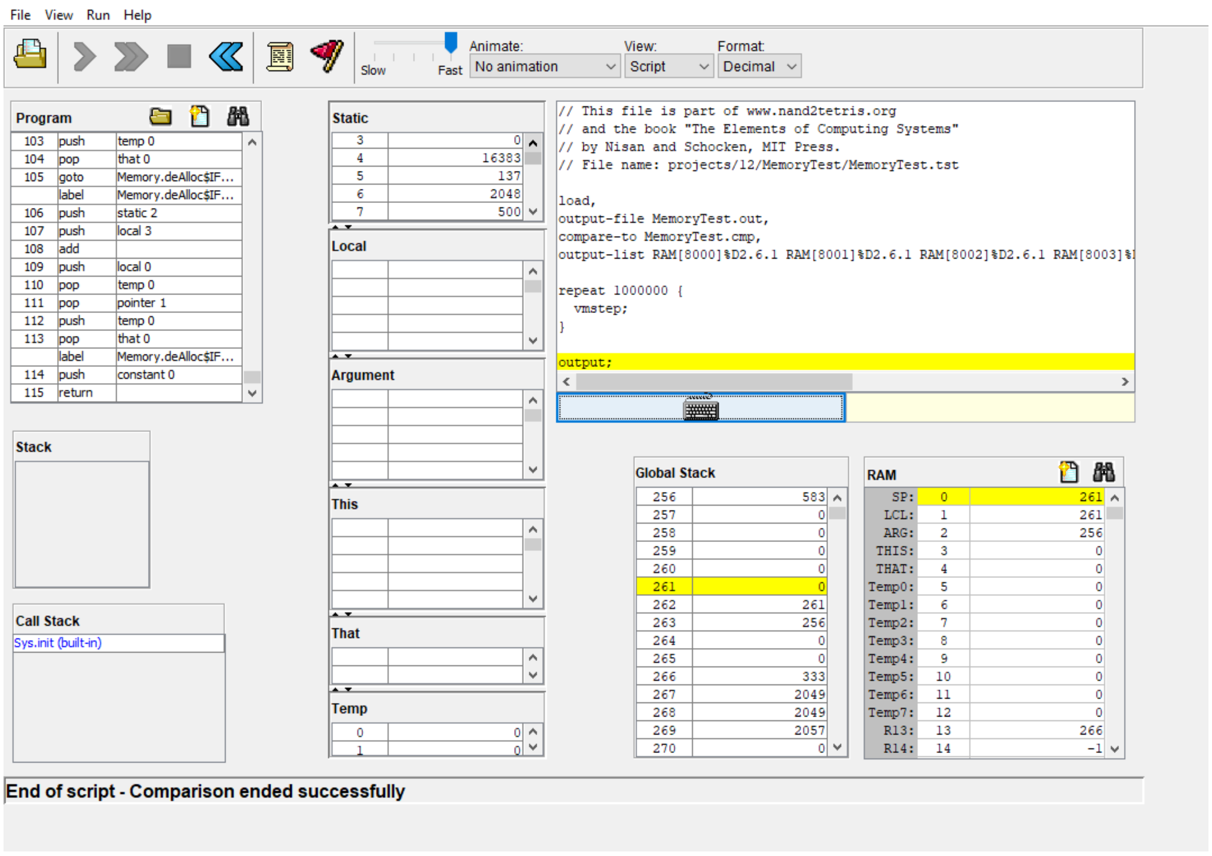
RAM ADDRESS	USAGE
0-15:	16-Virtual registers
16-225:	Static variables
256-2047:	Stack
2048-16483:	Heap
16384-24575:	Memory-mapped I/O

Code for Memory Allocations

```
class Memory {
      static int MEMORY, HEAP_BASE, NEXT_POINTER, SIZE,
HEAP_LIMIT, OOM, freeList, I;

  /** Initializes memory parameters. */
  function void init() {
    let MEMORY = 0;
    let SIZE = 0;
    let NEXT_POINTER = 1;
    let HEAP_BASE = 2048;
    let HEAP_LIMIT = 16383;
    let freeList = HEAP_BASE;
    let freeList[SIZE] = HEAP_LIMIT - HEAP_BASE;
    let OOM = 137;
    let i = 0;

    return;
  }

  /** Returns the value of the main memory at the given
address. */
  function int peek(int address) {
    return MEMORY[address];
  }

  /** Sets the value of the main memory at this address
   *  to the given value. */
  function void poke(int address, int value) {
    let MEMORY[address] = value;

    return;
  }

  /** finds and allocates from the heap a memory block
of the
   *  specified size and returns a reference to its base
address. */
  function int alloc(int size) {
    var bool defragged;
    var int currentFreeSegment, block, newFreeSegment,
previousFreeSegment;

    let defragged = false;
    let currentFreeSegment = freeList;

    while (~block) {
      if (currentFreeSegment[SIZE] > size) {
        if (currentFreeSegment = freeList) {
          if (currentFreeSegment[NEXT_POINTER]) {
            let freeList =
currentFreeSegment[NEXT_POINTER];
          } else {
            let freeList = currentFreeSegment + size +
1;
            let freeList[SIZE] =
currentFreeSegment[SIZE] - size - 1;
          }
        } else {
          if (size < (currentFreeSegment[SIZE] - 2)) {
            let newFreeSegment = currentFreeSegment +
size + 1;
            let newFreeSegment[SIZE] =
currentFreeSegment[SIZE] - size - 1;
            let previousFreeSegment[NEXT_POINTER] =
newFreeSegment;
          } else {
            let previousFreeSegment[NEXT_POINTER] =
currentFreeSegment[NEXT_POINTER];
          }
        }

        let block = currentFreeSegment + 1;
      } else {
        if (currentFreeSegment[NEXT_POINTER] = null) {
          if (defragged) {
            do Sys.error(OOM);
          } else {
            do Memory.defrag();
            let defragged = true;
            let currentFreeSegment = freeList;
          }
        } else {
          let previousFreeSegment = currentFreeSegment;
          let currentFreeSegment =
currentFreeSegment[NEXT_POINTER];
        }
      }
    }

    let block[-1] = size + 1;
    let block[0] = null;

    return block;
  }
  function void defrag() {
    var int currentFreeSegment, nextFreeSegment;

    let currentFreeSegment = freeList;
    let nextFreeSegment =
currentFreeSegment[NEXT_POINTER];

    while (~(nextFreeSegment = 0)) {
      if ((currentFreeSegment +
currentFreeSegment[SIZE]) = nextFreeSegment) {
        let currentFreeSegment[SIZE] =
currentFreeSegment[SIZE] + nextFreeSegment[SIZE];
        let currentFreeSegment[NEXT_POINTER] =
nextFreeSegment[NEXT_POINTER];
        let nextFreeSegment[SIZE] = null;
        let nextFreeSegment[NEXT_POINTER] = null;
      }

      let currentFreeSegment = nextFreeSegment;
      let nextFreeSegment =
currentFreeSegment[NEXT_POINTER];
    }

    return;
  }

  /** De-allocates the given object and frees its space.
*/
  function void deAlloc(int block) {
    var int segment, segmentLength, oldFreeList,
currentFreeSegment, oldNextSegment;

    let segment = block - 1;
    let segmentLength = segment[0];

    while (i < (segmentLength - 1)) {
      let block[i] = null;
      let i = i + 1;
    }

    if (segment < freeList) {
      let oldFreeList = freeList;
      let freeList = segment;
      let freeList[NEXT_POINTER] = oldFreeList;
      return;
    }

    let currentFreeSegment = freeList;

    while ((currentFreeSegment[NEXT_POINTER] < segment)
& currentFreeSegment[NEXT_POINTER]) {
      let currentFreeSegment =
currentFreeSegment[NEXT_POINTER];
    }

    if (currentFreeSegment[NEXT_POINTER] > 0) {
      let oldNextSegment =
currentFreeSegment[NEXT_POINTER];
      let currentFreeSegment[NEXT_POINTER] = segment;
      let segment[NEXT_POINTER] = oldNextSegment;
    } else {
      let currentFreeSegment[NEXT_POINTER] = segment;
    }

    return;
  }
}
```

Results

Simulation in VM Emulator



Animate: No animation
View: Script
Format: Decimal

Program

103	push	temp 0
104	pop	that 0
105	goto	Memory.deAlloc$IF...
	label	Memory.deAlloc$IF...
106	push	static 2
107	push	local 3
108	add	
109	push	local 0
110	pop	temp 0
111	pop	pointer 1
112	push	temp 0
113	pop	that 0
	label	Memory.deAlloc$IF...
114	push	constant 0
115	return	

Call Stack: Sys.init (built-in)

Static

3	0
4	16383
5	137
6	2048
7	500

Temp

0	0
1	0

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/12/MemoryTest/MemoryTest.tst

load,
output-file MemoryTest.out,
compare-to MemoryTest.cmp,
output-list RAM[8000]%D2.6.1 RAM[8001]%D2.6.1 RAM[8002]%D2.6.1 RAM[8003]%

repeat 1000000 {
  vmstep;
}

output;
```

Global Stack

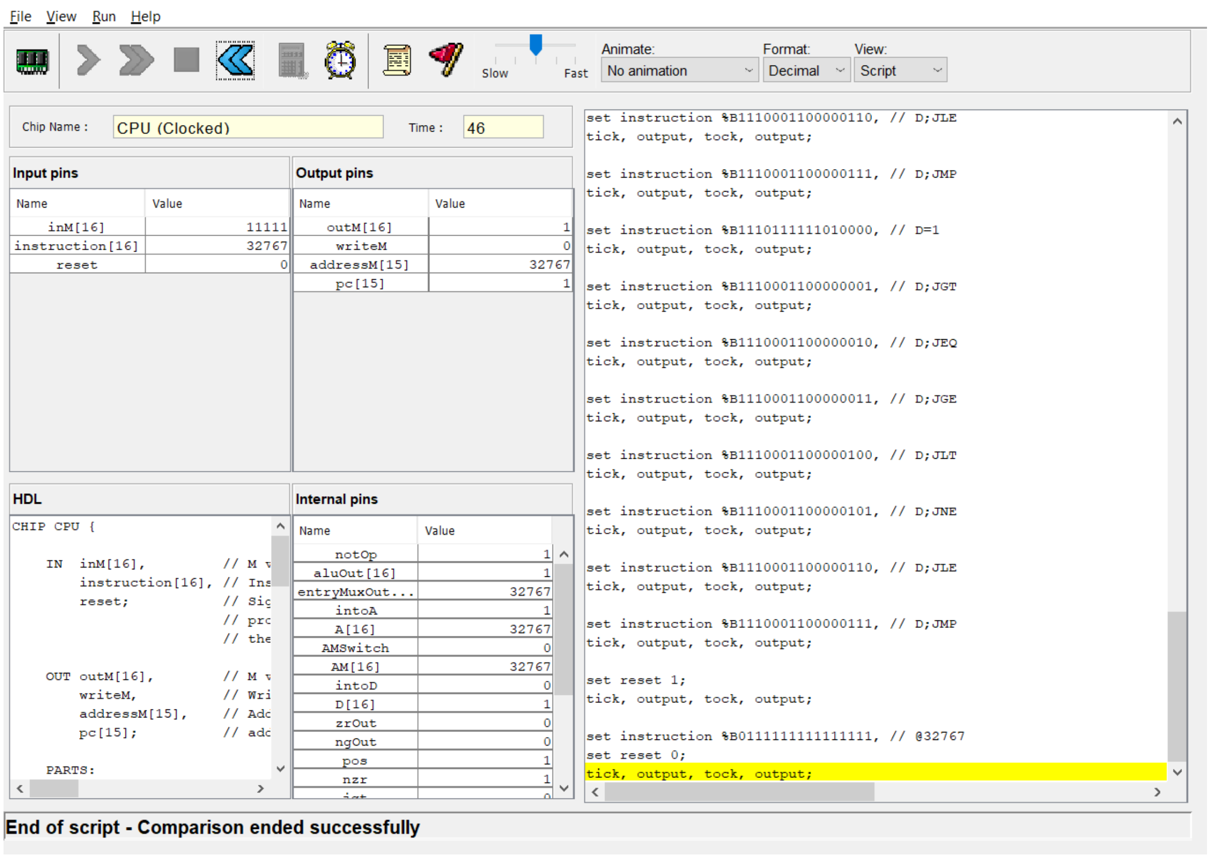
256	583
257	0
258	0
259	0
260	0
261	0
262	261
263	256
264	0
265	0
266	333
267	2049
268	2049
269	2057
270	0

RAM

SP:	0	261
LCL:	1	261
ARG:	2	256
THIS:	3	0
THAT:	4	0
Temp0:	5	0
Temp1:	6	0
Temp2:	7	0
Temp3:	8	0
Temp4:	9	0
Temp5:	10	0
Temp6:	11	0
Temp7:	12	0
R13:	13	266
R14:	14	-1

End of script - Comparison ended successfully

Simulation in Hardware simulator



Animate: No animation
Format: Decimal
View: Script

Chip Name : CPU (Clocked)   Time : 46

Input pins

Name	Value
inM[16]	11111
instruction[16]	32767
reset	0

Output pins

Name	Value
outM[16]	1
writeM	0
addressM[15]	32767
pc[15]	1

HDL

```
CHIP CPU {

    IN  inM[16],         // M v
        instruction[16], // Ins
        reset;           // Sig
                         // prc
                         // the

    OUT outM[16],        // M v
        writeM,          // Wri
        addressM[15],    // Adc
        pc[15];          // adc

    PARTS:
```

Internal pins

Name	Value
notOp	1
aluOut[16]	1
entryMuxOut...	32767
intoA	1
A[16]	32767
AMSwitch	0
AM[16]	32767
intoD	0
D[16]	1
zrOut	0
ngOut	0
pos	1
nzr	1

```
set instruction %B1110001100000110, // D;JLE
tick, output, tock, output;

set instruction %B1110001100000111, // D;JMP
tick, output, tock, output;

set instruction %B1110111111010000, // D=1
tick, output, tock, output;

set instruction %B1110001100000001, // D;JGT
tick, output, tock, output;

set instruction %B1110001100000010, // D;JEQ
tick, output, tock, output;

set instruction %B1110001100000011, // D;JGE
tick, output, tock, output;

set instruction %B1110001100000100, // D;JLT
tick, output, tock, output;

set instruction %B1110001100000101, // D;JNE
tick, output, tock, output;

set instruction %B1110001100000110, // D;JLE
tick, output, tock, output;

set instruction %B1110001100000111, // D;JMP
tick, output, tock, output;

set reset 1;
tick, output, tock, output;

set instruction %B0111111111111111, // @32767
set reset 0;
tick, output, tock, output;
```

End of script - Comparison ended successfully

Inference

Thus we can infer that after the execution of a program of memory. jack, stack, static and heap got implemented in between in their respective ram addresses and finally, we built the CPU with a greater ROM(32k) and we were able to run and get the output for the .asm programs.

Conclusion

Thus we have allocated the storage for heap , stack and static in jack programming and converted it into a VM file and converted to ASM to check the correctness of our code as well as the CPU implementation.

References

- Materials provided by Dr.Jyothish Lal Sir
- Some content from Coursera
- NPTEL videos suggested by Dr.Jyothish Lal Sir

Contributions

1.	Anirudh Edpuganti	- Jack Implementation of Memory allocation algorithm and report
2.	Rishekesan	- Implementation of Stack and Static and Report
3.	Shreya	- Theory of Heap and implementation of CPU
4.	Bhoomika	- CPU implementation and PPT
5.	Gokul	- CPU implementation and PPT