Ministry of Science and Education of Russian Federation

Peter the Great St.Petersburg Polytechnic University

—

Institute of Computer Science and Technology

**Department «Information security of computer systems»**

**LAB №5**

**Simulating the software stack**

course «OOP»

Student

Gr. 33508/3 Pavel Knyazev

<*signature*>

Instructor Andrey Y. Chernov

<*signature*>

Saint-Petersburg

2017

CONTENTS

1 Task 2

2 Introduction 2

3 Results 3

4 Conclusion 7

1. **Task**

Objective: to Develop a program on the Ogre engine which visualizes the operation of the software stack. The program should display the memory stack addresses and values in this memory area, a pointer ESP, EBP, the value of the EAX register, the memory area with program code, a pointer EIP. The program needs to read a file with the commands in XML format (you can use any existing library XML parser).

Minimum supported commands:

• mov eax, number

• mov esp, the number

• push the number

• pop eax

• mov esp, ebp

• mov ebp, esp

• push esp

• push ebp

• push eax

• pop ebp

• pop esp

• call eax

• call number

• jmp eax

• jmp number

• ret

The program needs for the event (e.g., pressing the space bar) to move from the current statement to the next with all the commands in the stack value should change the value of EAX must change, ESP, EBP, EIP should move pointing to specific locations in memory.

1. **Introduction**

In summary processor contains 12 software-accessible registers and the flags register (FLAGS) and the instruction pointer (IP). General purpose registers (RON) AX, BX, CX and DX are used to store data and perform different arithmetic and logical operations. In addition, each of these registers is divided into 2 parts for 8-bit, with which you can work with both 8-bit registers (AH, AL, BH, BL, CH, CL, DH, DL). The younger part of the registers are labeled with the letter L (Low), and senior H (High). Some commands implicitly use a certain register, for example, CX can act as a loop counter. Index registers are used for storing indexes when working with arrays. SI (Source Index) holds the index of the source, and DI (Destination Index) — the index of the receiver, although they can be used as General purpose registers. Registers-pointers BP and SP are used to working with stack. BP (Base Pointer) allows you to work with variables on the stack. It can also be used for other purposes. SP (Stack Pointer) indicates the top of the stack. It is used by commands that manipulate the stack. (About a stack I will detail in a separate part of the curriculum). Segment registers CS (Code Segment), DS (Data Segment), SS (Stack Segment) and ES (Enhanced Segment) is designed to provide a segmented addressing. The code is in the code segment, data in the data segment, the stack in the stack segment and extra segment data. Actual physical address is acquired by shifting the contents of a segment register by 4 bits to the left and adding to it the offset (relative address within the segment). Read more about segmented addressing in part 31.

COM the program is always in one segment, which is also the code segment, data and stack. When you run a COM program segment registers will contain identical values. The instruction pointer IP (Instruction Pointer) contains the address (in the code segment). Directly modify its contents impossible, but the CPU does it himself. When you execute a normal command value IP is incremented by the size of the executed command. There are also commands the transfer of control, which change the value of the IP to navigate within the program.

1. **Results**

**Building OGRE.**

OGRE sources - https://bitbucket.org/sinbad/ogre/downloads/

OGRE dependencies - https://bitbucket.org/cabalistic/ogredeps

Steps:

1. Compile dependencies
   1. Download sources of OGRE dependencies
   2. Generate VS project via CMake
   3. Build libraries in VS project
2. Compile OGRE
   1. In CMake add path to dependencies build folder
   2. Generate VS project
   3. Build VS project
3. Using OGRE
   1. In new VS project add paths to libraries, headers to OGRE sdk which was built.
   2. Add .dll files to executable folder
   3. Run

**Implemented stack**

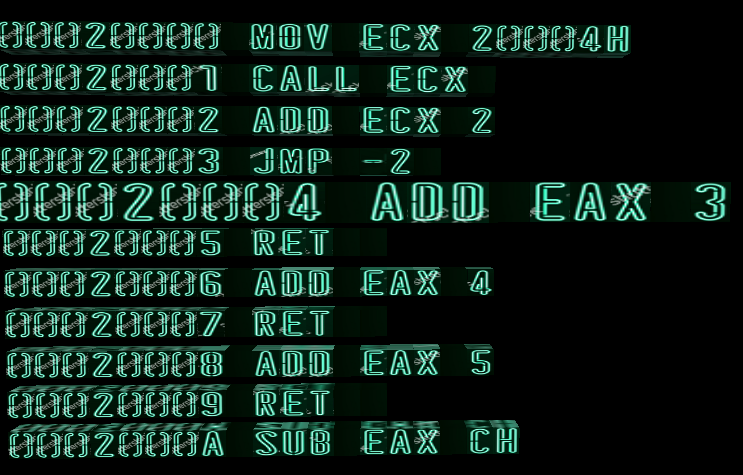
The program was written in C++. To visualize the stack was used OGRE.

Program instructions: ( r – register, d – number )

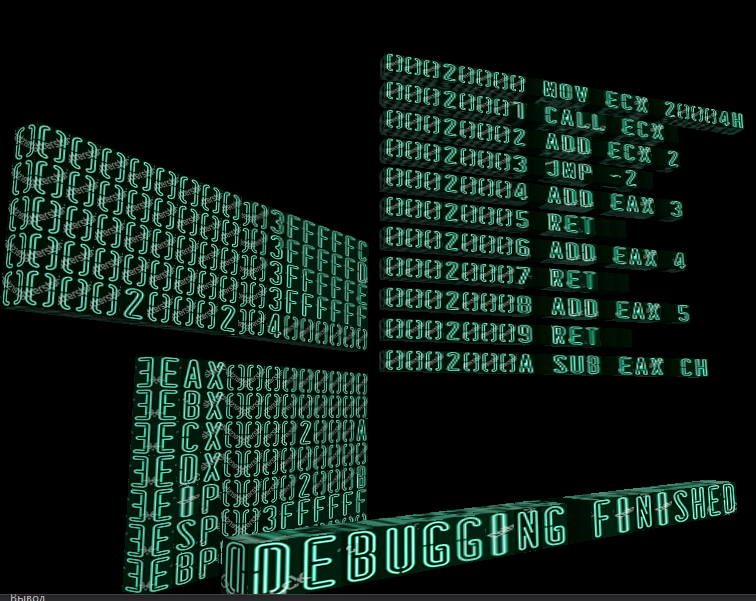
* mov r1, r2; mov r1, d
* add r1, r2; add r1, d
* sub r1, r2; sub r1, d
* push r; push d
* pop r
* call r; call d
* jmp r; jmp d
* ret
* retn d

When the xml file gets unsupported instruction manual, you will get an error message

In figures 1, 2 the results of the program are shown.



Pic. 1 – Executing instructions



Pic. 2 – Debugging finished

**Stack Primitives**

All elements of the program consist of objects CubeLetter, CubeLetter Class is responsible for rendering a single character numbers or letters. Multiple instances CubeLetter contained in the class CubeText. CubeText is responsible for rendering a single word. Classes CubeRegister, CubeInstruction, CubeStack contain object instances CubeText to visualize their content.

**Description of stack work**

When the object is created CubeStack following parameters are set: the stack size, the starting position for the esp register, its position in space. Stack directly vzaimodeistvuet with the esp register, the value of which can be changed from outside or inside the class constructor creates visual objects – addresses to which to write the data, and the memory initialized to zero. In the push operation writes new data to the buffer and update visual content. Next, the esp register is decremented by 1 If esp is outside of the allocated stack memory, the operation fails. In the operation pop the esp register is added 1, and its data is not deleted. To obtain the value on top of the stack can be used the method topVal.

**Description of register work**

Each register is described by a class CubeRegister. In this class there is an object CubeText to visualize the content and the value field to store the value of the register. Also, the register has methods setValue, getValue, Allowing you to modify its contents.

To store all the registers using a class CubeRegisterList that contains container std::map. Key for container – the name of the register (string) value pointer to the found register. If the requested register is not in container, it returns null.

The container registers: eax, ebx, ecx, edx, eip, esp, ebp.

**Description of InstructionStorage class.**

This class is used to store instructions, visualisation, parsing the xml file. When you create it parses the xml file and stores the xml of the user in the desired format in a vector container. After that it is possible to setup data and instructions. Its main method getInstruction(uint i) where I is the number of instructions starting from the beginning. If the given index is empty, it returns null, otherwise the context of the user: command, argument 1, argument 2 (if any).

**Description of AsmCommand class.**

This class is a base relative to the other classes to perform operations mov, add, e.t.c. It has 2 virtual methods: exec (request the required registers/values and execute the statement), setupESP (change the eip value after execution). For each of the operations these methods can be different or the same Vice versa. From this class the following classes are inherited: AsmMove, AsmCall, AsmJmp ... e.t.c. Each of them performs its operation.

At the entrance of the exec method serves: list of registers, parameter values, pointer to the stack. First, there is the query of the registers r1, r2 in the parameters d1, d2 (which may be a number). If register is not found, it comes back null if the parameters are in an incorrect format (d == null), the function will return an error message.

**A description of each of the methods in the inherited classes: (pseudo C – style)**

* AsmMove: r1 = (r2 == null) ? (d == null) ? error : d : r2;
* AsmAdd: r1 = (r2 == null) ? (d == null) ? error : r1 + d : r1 + r2;
* AsmSub: r1 = (r2 == null) ? (d == null) ? error : r1 - d : r1 - r2;
* AsmPush: value = (r1 == null) ? (d == null) ? error : d : r1; Stack.push(value); esp--;
* AsmPush: (r1 == null) ? error : r1 = stack.top(); esp++;
* AsmCall: eip = (r1 == null) ? (d ==null) ? error : d : r; stack.push(eip + 1); esp--;
* AsmJmp: eip = (r1 == null) ? (d ==null) ? error : d : r;
* AsmRet: eip = stack.top(); esp++;
* AsmRetn: n = (d == null) ? error : d; esp+=n eip = stack.top(); esp++;

**Description of InstructionWalker class.**

An object of this class stores pointers to a list of registers, stack, and store instructions. Primary user manual – execCom. It is the fulfillment of all instructions of the program. This happens in the following way:

1. Get current instruction address (via eip register)
2. Get current instruction context
3. Unmark previous instruction
4. Get pointer to AsmCommand class (queryCommand)
5. Call exec method of that class
6. Call setupEIP method of that class
7. Check the eip address is valid (getInstruction(address))
8. Move instruction list
9. Mark next instruction

**Description of input xml – file format**

For parsing xml – file the RapidXml parser was used.

The first three lines contain **setup** nodes. Setup nodes are needed to set initial setting: program start address, stack pointer address, stack size. Next nodes named **command** contain parameters containing command name: mov, add, call, e.t.c. These nodes contain subnodes named **argument** containing parameters value, which can be the name of the register of number.

Below is the example of xml-file.

<setup org\_address="20000h"/>

<setup esp\_address="4000000h"/>

<setup stack\_size="5"/>

<command name="mov"> <argument value="ecx"/> <argument value="20004h"/> </command>

<command name="call"> <argument value="ecx"/> </command>

1. **Conclusion**

During this task basics of the graphics engine Ogre3d have been learned. 3D-modeling skills have been also obtained. This lab has allowed to apply in practice knowledge in OOP. This lab also allows to create a classes dependency hierarchy.

Attachment

Source.cpp

//////////////////////// asm commands

class AsmCommand

{

protected:

CubeRegister\* reg1;

CubeRegister\* reg2;

uint toHex(string argStr)

{

int radix = (argStr.back() == 'h' || argStr.back() == 'H') ? 16 : 10;

return stoi(argStr, NULL, radix);

}

void getRegisters(CubeRegisterList\* rList, const string& argStr1, const string& argStr2)

{

reg1 = rList->queryRegister(argStr1);

reg2 = rList->queryRegister(argStr2);

}

public:

virtual string exec(CubeRegisterList\* rList, CubeStack\* stack,

const string& argStr1, const string& argStr2)

{

return "Ok";

}

virtual uint setupEIP(CubeRegister\* eip, bool\* reverse)

{

uint val = eip->getValue();

eip->set\_value(val + 1);

\*reverse = false;

return Cubic::delta\_y + Cubic::size\_y;

}

};

////////////////////////////////// imlemented asm commands

// mov r1, r2 ; mov r1, d

class AsmMove : public AsmCommand

{

string exec(CubeRegisterList\* rList, CubeStack\* stack,

const string& argStr1, const string& argStr2) override

{

getRegisters(rList, argStr1, argStr2);

if (reg1 != NULL && reg2 != NULL)

{

uint val = reg2->getValue();

reg1->set\_value(val);

return "Ok";

}

else if (reg1 != NULL && argStr2 != "")

{

uint val = toHex(argStr2);

reg1->set\_value(val);

return "Ok";

}

else

return "command mov invalid usage";

}

};

// add r1, r2; add r1, d

class AsmAdd : public AsmCommand

{

string exec(CubeRegisterList\* rList, CubeStack\* stack,

const string& argStr1, const string& argStr2) override

{

getRegisters(rList, argStr1, argStr2);

if (reg1 != NULL && reg2 != NULL)

{

uint val = reg2->getValue();

reg1->set\_value(reg1->getValue() + val);

return "Ok";

}

else if (reg1 != NULL && argStr2 != "")

{

uint val = toHex(argStr2);

reg1->set\_value(reg1->getValue() + val);

return "Ok";

}

else

return "command mov invalid usage";

}

};

// sub r1, r2; sub r1, d

class AsmSub : public AsmCommand

{

string exec(CubeRegisterList\* rList, CubeStack\* stack,

const string& argStr1, const string& argStr2) override

{

getRegisters(rList, argStr1, argStr2);

if (reg1 != NULL && reg2 != NULL)

{

uint val = reg2->getValue();

reg1->set\_value(reg1->getValue() - val);

return "Ok";

}

else if (reg1 != NULL && argStr2 != "")

{

uint val = toHex(argStr2);

reg1->set\_value(reg1->getValue() - val);

return "Ok";

}

else

return "command mov invalid usage";

}

};

// push r; push d

class AsmPush : public AsmCommand

{

string exec(CubeRegisterList\* rList, CubeStack\* stack,

const string& argStr1, const string& argStr2) override

{

getRegisters(rList, argStr1, argStr2);

if (reg1 != NULL || argStr1 != "")

{

uint val;

if (reg1 != NULL)

val = reg1->getValue();

else

val = toHex(argStr1);

if (!stack->push(val))

return "Stack overflow";

return "Ok";

}

else

return "command push invalid usage";

}

};

// pop r

class AsmPop : public AsmCommand

{

string exec(CubeRegisterList\* rList, CubeStack\* stack,

const string& argStr1, const string& argStr2) override

{

getRegisters(rList, argStr1, argStr2);

if (reg1 != NULL)

{

uint val = stack->topVal();

if (!stack->pop())

return "Stack underflow";

reg1->set\_value(val);

return "Ok";

}

else

return "command pop invalid usage";

}

};

// call r; call d

class AsmCall : public AsmCommand

{

uint newPos = 0;

string exec(CubeRegisterList\* rList, CubeStack\* stack,

const string& argStr1, const string& argStr2) override

{

getRegisters(rList, argStr1, argStr2);

if (reg1 != NULL || argStr1 != "")

{

if (reg1 != NULL)

newPos = reg1->getValue();

else

newPos = toHex(argStr1);

// push to stack addr of next instruction

CubeRegister\* eip = rList->queryRegister("eip");

if (!stack->push(eip->getValue() + 1))

return "Stack overflow";

return "Ok";

}

else

return "command call invalid usage";

}

virtual uint setupEIP(CubeRegister\* eip, bool\* rev) override

{

uint curPos = eip->getValue();

eip->set\_value(newPos);

uint delta;

if (newPos > curPos)

{

\*rev = false;

delta = newPos - curPos;

}

else

{

\*rev = true;

delta = curPos - newPos;

}

return (Cubic::delta\_y + Cubic::size\_y) \* delta;

}

};

// jmp r; jmp d

class AsmJmp : public AsmCommand

{

uint newPos = 0;

string exec(CubeRegisterList\* rList, CubeStack\* stack,

const string& argStr1, const string& argStr2) override

{

getRegisters(rList, argStr1, argStr2);

uint delta;

if (reg1 != NULL || argStr1 != "")

{

if (reg1 != NULL)

delta = reg1->getValue();

else

delta = toHex(argStr1);

CubeRegister\* eip = rList->queryRegister("eip");

newPos = delta + eip->getValue();

return "Ok";

}

else

return "command jmp invalid usage";

}

virtual uint setupEIP(CubeRegister\* eip, bool\* rev) override

{

uint curPos = eip->getValue();

eip->set\_value(newPos);

uint delta;

if (newPos > curPos)

{

\*rev = false;

delta = newPos - curPos;

}

else

{

\*rev = true;

delta = curPos - newPos;

}

return (Cubic::delta\_y + Cubic::size\_y) \* delta;

}

};

// ret

class AsmRet : public AsmCommand

{

uint newPos = 0;

string exec(CubeRegisterList\* rList, CubeStack\* stack,

const string& argStr1, const string& argStr2) override

{

// get return address

newPos = stack->topVal();

if (!stack->pop())

return "Stack underflow";

return "Ok";

}

virtual uint setupEIP(CubeRegister\* eip, bool\* rev) override

{

uint curPos = eip->getValue();

eip->set\_value(newPos);

uint delta;

if (newPos > curPos)

{

\*rev = false;

delta = newPos - curPos;

}

else

{

\*rev = true;

delta = curPos - newPos;

}

return (Cubic::delta\_y + Cubic::size\_y) \* delta;

}

};

// retn d

class AsmRetn : public AsmCommand

{

uint newPos = 0;

string exec(CubeRegisterList\* rList, CubeStack\* stack,

const string& argStr1, const string& argStr2) override

{

if (argStr1 == "")

return "Command retn invalid usage";

// pop few bytes in stack

uint valToPop = stoi(argStr1);

if (valToPop % 4 != 0) // 4 bytes

return "Command retn invalid parameter " + uintValToHexString(valToPop);

CubeRegister\* esp = rList->queryRegister("esp");

esp->set\_value(esp->getValue() + valToPop / 4);

// get return address

newPos = stack->topVal();

if (!stack->pop())

return "Stack underflow";

return "Ok";

}

virtual uint setupEIP(CubeRegister\* eip, bool\* rev) override

{

uint curPos = eip->getValue();

eip->set\_value(newPos);

uint delta;

if (newPos > curPos)

{

\*rev = false;

delta = newPos - curPos;

}

else

{

\*rev = true;

delta = curPos - newPos;

}

return (Cubic::delta\_y + Cubic::size\_y) \* delta;

}

};

////////////////////////////////// imlemented asm commands

class AsmCommandList

{

std::map<string, AsmCommand\*> comMap;

public:

AsmCommandList()

{

AsmCommand\* mov = new AsmMove();

comMap.insert(pair<string, AsmCommand\*>("mov", new AsmMove()));

comMap.insert(pair<string, AsmCommand\*>("push", new AsmPush()));

comMap.insert(pair<string, AsmCommand\*>("pop", new AsmPop()));

comMap.insert(pair<string, AsmCommand\*>("call", new AsmCall()));

comMap.insert(pair<string, AsmCommand\*>("jmp", new AsmJmp()));

comMap.insert(pair<string, AsmCommand\*>("ret", new AsmRet()));

comMap.insert(pair<string, AsmCommand\*>("retn", new AsmRetn()));

comMap.insert(pair<string, AsmCommand\*>("add", new AsmAdd()));

comMap.insert(pair<string, AsmCommand\*>("sub", new AsmSub()));

comMap.insert(pair<string, AsmCommand\*>("nop", new AsmCommand()));

}

~AsmCommandList()

{

for (auto& elem : comMap)

{

delete elem.second;

}

}

AsmCommand\* queryCommand(string com)

{

auto x = comMap.find(com);

if (x != comMap.end())

return x->second;

return NULL;

}

};

/////////////////////// asm commands

class InstructionWalker

{

AsmCommandList\* cList;

CubeRegisterList\* rList;

CubeStack\* pStack;

CubeInstructionStorage\* iStorage;

///// graphics

void unmarkInstruction(uint addr)

{

SceneNode\* xNode = iStorage->getInstruction(addr)->getSceneNode();

Vector3 scale = xNode->getScale();

xNode->setScale(scale / 1.5);

}

void markInstruction(uint addr)

{

SceneNode\* xNode = iStorage->getInstruction(addr)->getSceneNode();

Vector3 scale = xNode->getScale();

xNode->setScale(scale \* 1.5);

}

void move(uint val, bool reverse)

{

Real delta = (Real)val;

if (reverse)

delta \*= -1;

iStorage->getNode()->translate(Vector3(0.0f, delta, 0.0f));

}

public:

InstructionWalker(

AsmCommandList\* cList,

CubeRegisterList\* rList,

CubeStack\* pStack,

CubeInstructionStorage\* iStorage

) : cList(cList), rList(rList), iStorage(iStorage), pStack(pStack)

{

CubeRegister\* eip = rList->queryRegister("eip");

eip->set\_value(iStorage->getOrg());

markInstruction(iStorage->getOrg());

}

///// exec

string execCom()

{

string comName, r1, r2;

// get cur instruction address

CubeRegister\* eip = rList->queryRegister("eip");

uint addr = eip->getValue();

// get instruction context

CubeInstruction\* x = iStorage->getInstruction(addr);

InstructionContext con = x->getInstructionContext();

comName = con.command;

r1 = con.arg1;

r2 = con.arg2;

// unmark current instruction

unmarkInstruction(addr);

// execute current instruction

AsmCommand\* asmCom = cList->queryCommand(comName);

if (asmCom == NULL)

return "Unsupported instruction " + comName + " on address " + uintValToHexString(addr) + "H";

string stat = asmCom->exec(rList, pStack, r1, r2);

if (stat != "Ok")

return "Execute error on address "+ uintValToHexString(addr) + "H" + " " + stat;

// change eip value

// rev indicates is jump up or down

bool rev;

uint valToMove = asmCom->setupEIP(eip, &rev);

// before moving check if invalid address was passed (jmp call)

uint uAddr = eip->getValue();

if (iStorage->getInstruction(uAddr) == NULL)

{

// check if it's last instruction

if (iStorage->isLastInstruction(uAddr))

return "Debugging finished";

return "Access Violation on address " + uintValToHexString(uAddr) + "H";

}

// move list

move(valToMove, rev);

// mark current instruction

markInstruction(eip->getValue());

return "Ok";

}

};