Announcements

- Nothing this time.

Assignments

- Assignment 4 part 1 is out since Wednesday
  - Part 1 deadline: 20:00, Wed. Mar 7th
    - Contains assembly coding task
    - Please deliver one .s file, and your PDF
  - Part 2 out today, deadline 20:00, Wed. Mar 14th
  - Topics: Stack frames, x86 assembly, symbol tables

TDT4205 Compilers

Stack Frames And Assembly Programming

Table of Content

- x86 assembly
  - Crash course in x86 assembly, Intel syntax
  - How to code, compile and run
  - The stack
- Stack Frames
  - How will they look like in VSL?
- PS4 part 2, info

x86 assembly

- There are many variants of the assembly programming language for x86
  - Two main variants: Intel and AT&T
- We will be using the Intel syntax.

Quick notes on syntax

- Constants: Prepend with $. So the constant 5 is written $5.
    - Example: “push $20” => push 20 on the stack
- Registers: Prepend with %. So the EAX register is referred to as %eax
    - Example: “addl $5, %eax” => add 5 to EAX

Quick notes on syntax (assembly has pointers too!)

- Dereference the address in a register: Enclose the register in (%reg).
  - Example: addl $5,(%esp)
    - Add 5 to the value that's on top of stack
    - C-equivalent: *topofstack += 5;
- Add-and-dereference in one line (array lookup): N(%reg) ; N in bytes
  - Example: addl $5, 4(%esp)
  - Add 5 to the element next-to-the-top of stack(“tos[-1]”+=5)

A (very) simple example

```
.data
HELLO: .string "Hello World! Here's some integer: %d! \n"

.text
.globl main

main:
    push $8
    push $HELLO
    call printf
    add $8, %esp
    push $0
    call exit
```

Simple assembly

- Assembly programs has two main parts:
  - A .data segment. This is where your constants and other data go, like debugging information and such.
    - In VSL we put all string literals in here.
  - A .text segment. This is where your code go.

What happens?

```
.data
HELLO: .string "Hello World! Here's some integer: %d! \n"
```

Define a string constant in the data segment,
label it “HELLO”, containing the string we want to print

What happens? – cont'd

```
.text
.globl main
```

- Define the text segment
- .globl main: makes the function “main” visible to the linker. Important, since “main” is, as in C, the first function to be called
  - If the linker can't find it, it'll complain
    - Try it.

What happens? – cont'd

```
main:
    push $8
    push $HELLO
    call printf
    add $8, %esp
```

Equivalent in C:
int main() { printf(HELLO, 8); ... }
where HELLO is the string containing the formatting defined earlier

- Push two arguments on the stack, in *reverse order:* The number 8, and the format string
- Call printf. printf will get its arguments from the top of stack
- “Reset” the stack pointer

What happens? – cont'd

```
push $0
call exit
```

Equivalent in C:
int main() {… exit(0); }

- Push the constant 0 on the stack
- Call exit, to exit the application

Assigning to registers/mem locations

- Instructions:
  - movl A, B: set B to the value (of) A
    - movl $5, %eax sets the value of eax to 5
    - movl %eax, 4(%esp) sets the second-to-top item in the stack to the value of eax
    - movl 8(%ebp), %eax
      - On a low level, this sets eax to the value stored at the memory location %ebp+8.
      - On a higher level, the value of the first argument of the function is put into eax

Arithmetics

- addl A, B: Adds A to B (stores in B)
  - “addl $5, %eax” == eax += 5
  - “addl %eax, %ebx” == ebx += eax
- subl A, B: Subtracts A from B (stores in B)
  - “subl $5, %eax” == eax -= 5
- incl A, decl A: increment or decrement A with one
  - “incl %eax” == eax++

Arithmetics – cont'd

- Multiplication is trickier
- imul A
  - EDX:EAX ← EAX*A
  - A MUST be a register or memory location
    - imul $5 is illegal
    - imul %ebx or imul (%esp) is legal
  - Results are 64 bits; concatenated EDX with EAX

Arithmetics – cont'd

- Division is even trickier
- idiv A
  - EAX ← int(EDX:EAX / A)
  - EDX ← EDX:EAX mod A
  - A MUST be a register or memory location
    - idiv $5 is illegal
    - idiv %ebx or idiv (%esp) is legal
- Must sign-extend EAX into EDX if only using EAX

Arithmetics – cont'd

- Division is tricky
- Must sign-extend EAX into EDX if only using EAX
  - Otherwise EDX may contain rubbish
- Use the instruction cdq

```
movl $3, %ebx
movl $10, %eax
cdq
idiv %ebx
```

Divides 10 by 3.
After division:
EAX == 3, EDX == 1

Stack manipulation

- push A: Pushes the value of A to the stack
  - Semantically the same as
    - subl 4, %esp
    - movl A, (%esp)
- pop A: Pops the top-of-stack value into A
  - Semantically the same as
    - movl (%esp), A
    - addl $4, %esp

Comparison

- cmp A, B
  - Compares A with B. This sets a bunch of flags
  - We can perform jumps based on these flags (next slide)
  - B is the left-hand-side, A is the right-hand-side (Intel's engineers are clearly nuts :))

Jumps

- After comparing the operands, we can perform a “JUMP” to a label
  - Think of it as “if (condition) goto label1;”
- Conditional jumps (syntax of all are “jXX LABEL”):
  - je (jne) (jump if (not) equal)
  - jge/jle (jump if greater/less or equal)
  - jg/jl (jump if greater/less)

Jumps – cont'd

- There are also unconditional jumps:
  - jmp LABEL
    - doesn't care about status flags. Simply a “goto”
- Some instructions other than cmp can set status flags too
  - E.g. subl A,B:
    - jz LABEL jumps if B becomes 0
    - jnz LABEL jumps if B does NOT become 0

Loops

- There are no loops in assembly
  - Implemented with labels and jumps

C:

```
for (int i = 0; i < 10; i++)
  foo(i);
```

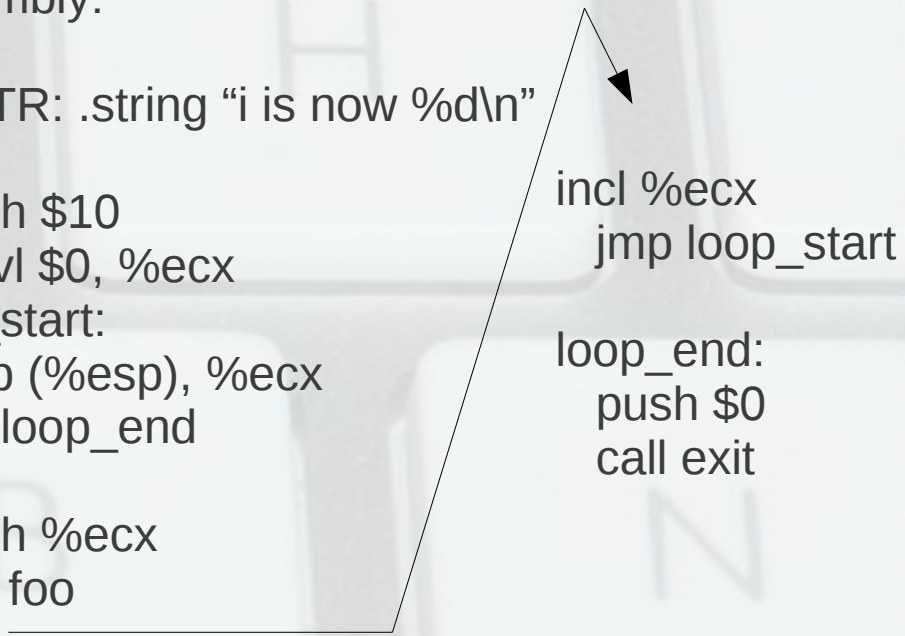
Assembly:

```
.data
MYSTR: .string “i is now %d\n”
…
   push $10
   movl $0, %ecx
loop_start:
   cmp (%esp), %ecx
   jge loop_end

   push %ecx
   call foo
```

```
incl %ecx
   jmp loop_start

loop_end:
   push $0
   call exit
```



Larger example - Fibonacci

- (demo)

Saving registers

- Most (all?) functions will use registers
- Registers are shared
- You can assume that if you call a function, the values in the registers that you dearly need gets destroyed
    - You can, however, assume that functions preserve the stack/base pointer
- We must save the registers

Saving registers – Cont'd

- Example:

```
.data
MYINT: .string "EAX is now: %d! \n"
...
    movl $314, %eax

    push %eax
    push $MYINT
    call printf
    add $8, %esp

    push %eax
    push $MYINT
    call printf
    add $8, %esp
```

Output:
EAX is now: 314!
EAX is now: 18!

Saving registers – Cont'd

- Solution: Push the registers onto the stack before calling the function

```
.data
MYINT: .string "EAX is now: %d! \n"
...
    movl $314, %eax

    push %eax
    push $MYINT
    call printf
    add $4, %esp /* only skip past the string pointer */
    pop %eax     /* restore eax */
```

Output:
EAX is now: 314!
EAX is now: 18!

Note: You should push all the registers your function/program depends on. Not only the ones that you send in as arguments to the function.

Stack frames

- Problem: No matter how deep the recursion is,
  - Your functions needs to know where to get its arguments
  - Your functions need to know where to get its local variables
  - Your function should preserve the stack, so that after your function returns, the stack is the same as before it was called

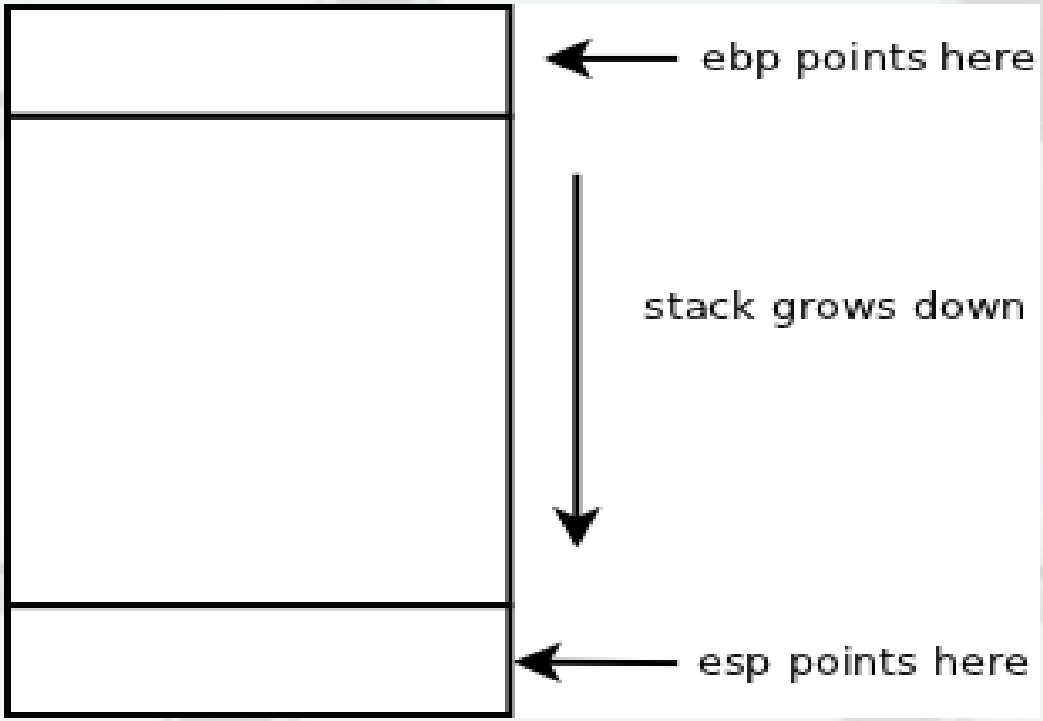
Stack frames – Cont'd

- Technicalities:
  - The register ESP points to the top of the stack
  - The register EBP poitns to the BASE of the current stack frame (“bottom of stack”, as seen from the current stack frame)

Stack frames – Cont'd

- Technicalities:
  - The register ESP points to the top of the stack
  - The register EBP poitns to the BASE of the current stack frame (“bottom of stack”, as seen from the current stack frame)

Stack frames – Cont'd



Stack frames – Cont'd

- To set up a stack frame:
  - Push old base pointer, ebp, onto stack
  - Set base pointer, ebp, to esp
    - we set the current top-of-stack (as it is in the beginning of the function call) to be the new base of our stack

Stack frames – Cont'd

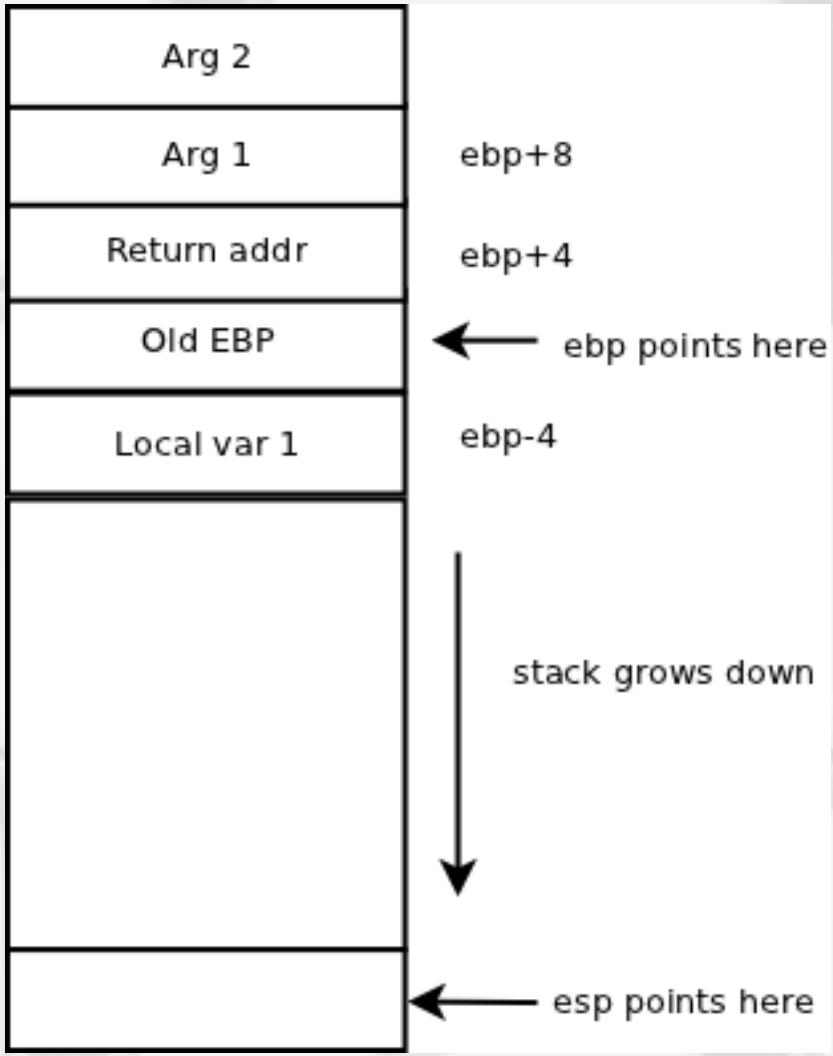
- To tear down a stack frame (i.e. put the state of the stack pointers back to how it was):
  - The current base pointer points to the element above the previous one
    - Set esp to ebp
    - Pop into ebp to restore the previous base pointer

Stack frames – cont'd

- After setting up a stack frame, ebp points to the bottom of the stack
  - Local variables are pushed onto the stack AFTER the stack frame is set up
  - Function arguments are pushed onto the stack BEFORE the stack frame is set up

Stack frames – cont'd

- Local variables are pushed onto the stack AFTER the stack frame is set up
  - We can reference them by offsetting from ebp. First local variable is at -4(ebp). Second at -8(ebp), …
- Function arguments are pushed onto the stack BEFORE the stack frame is set up
  - First argument is at 8(ebp), second at 12(ebp), ...


Arg 2
Arg 1
Return addr
Old EBP
Local var 1
ebp+8
ebp+4
ebp points here
ebp-4
stack grows down
esp points here

PS4, Part 2

- In part 2 of PS4 you will populate the symbol table with symbols
- You should use the library libghthash for your hash table
- Your task: Implement insertion and lookup into symbol tables, adding and removing scopes. Also, binding the names (variables) to symbols in the symbol table.

libghthash

- To use: #include <ght_hash_table.h>
  - Done in symtab.h
- To create a new hash table:
- `hash_t* tab = ght_create(N) (N is # of hash buckets)`
- Lookup:
- `value = ght_get(tab, strlen(key)+1, key);`
- Inserting:
- `ght_insert(tab, value, strlen(key)+1, key);`

Questions?

Appendix: Compiling assembly

- You can use the GNU assembler, “as”

```
as --32 myprogram.s -o myobjects.o //compile
gcc -m32 myobjects.o -o myprogram  // link
```

- Alternatively: Use gcc (which calls “as”)

```
gcc -m32 myprogram.s -o myprogram //compile and link
```

Appendix: Creating functions

- Functions are simply labels
- Call them with “call LABEL”
- Return to caller with “ret”

```
foo:
  /* do some stuff */
  Ret

main:
  …
  call foo
  …
```