Program Analysis

Lecture 04: *Machine Language III*
Winter term 2011/2012

Prof. Thorsten Holz

Announcements

- Feedback for tool intro and first exercise?
  - Next exercise will be published next week
  - Subroutines, code analysis, CrackMe, ...
- Next Wednesday: talk by @vxradius and @matrosov
  - See http://www.nds.rub.de/teaching/lectures/471/

Last Week

- x86 registers
- x86 instruction set
- Intel vs. AT&T syntax
- x86 memory access / endianness
- Privileges
- Interrupts / exceptions

Outline

- x86 subroutines
  - Overview
  - Calling conventions
- Higher-level structures
  - Control-flow structures
  - Loops

x86 Subroutines

Stack

- Memory area dedicated to local variables and calling information for subroutines
  - Meta data when calling functions
  - Synchronization (threads)
- Stack pointer `esp` indicates current top of stack
- Control stack with `push` and `pop` instructions

Base/Frame Pointer

- In addition to stack pointer there is the so called *base pointer/frame pointer* ebp
- Before modifying the stack pointer to allocate space for local variables during the prolog, the content of this register is saved in base pointer ebp
- Base pointer is used to adress local variables and parameters
- Not always used, so called *frame pointer omission* (FPO)

Subroutines

- Parameters are put on stack before function is called
  - Use `push` instruction
  - **Last parameter** is pushed **first**!

f(1, 2, 3)

```
push 3
push 2
push 1
call f
```

Subroutines

- Parameters are put on stack before function is called
  - Use `push` instruction
  - **Last parameter** is pushed **first**!

f(1, 2, 3)

`push 3`
`push 2`
`push 1`
`call f`

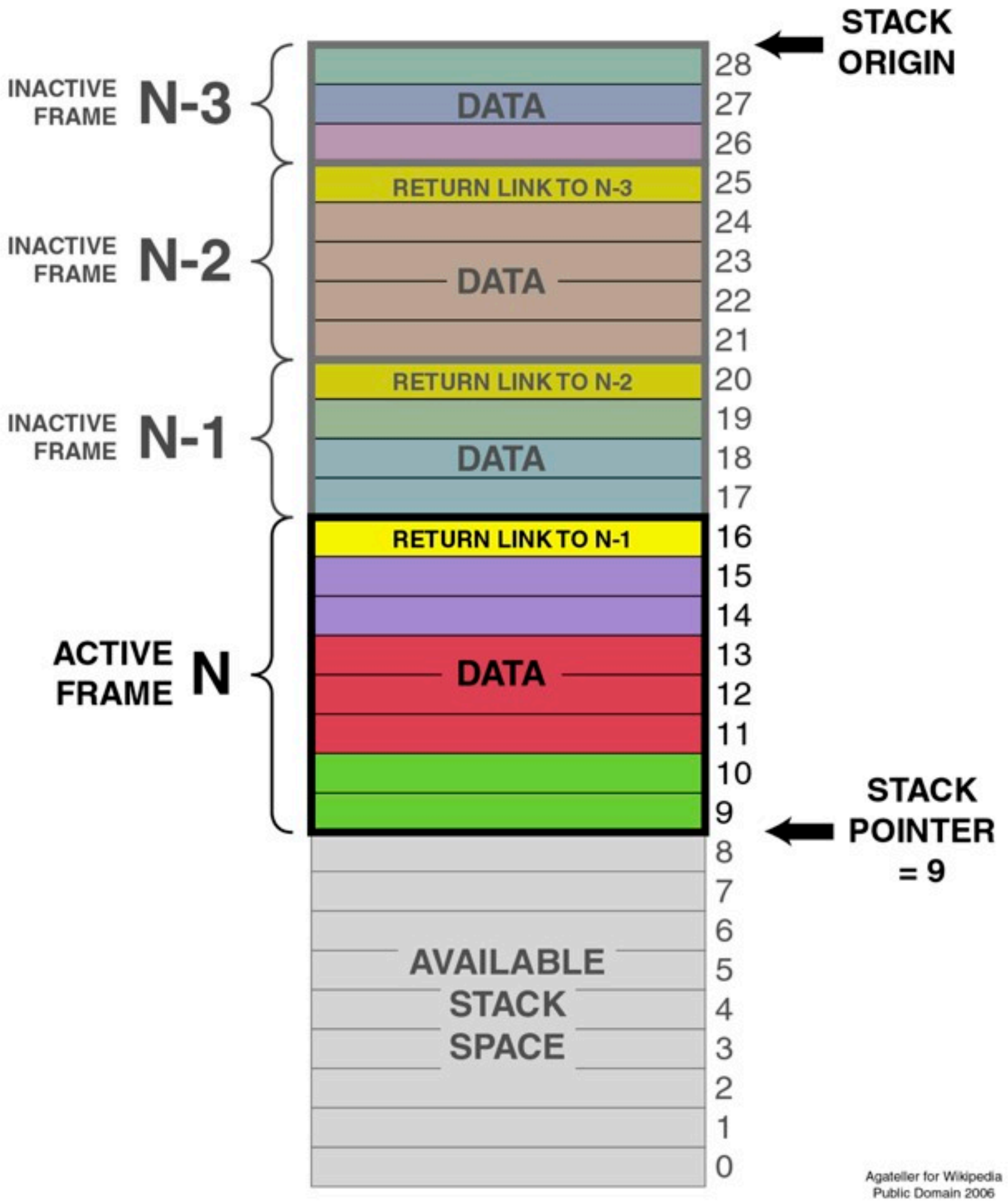
Function f

`push ebp       ; save old ebp`
`mov ebp, esp   ; load ebp with esp`
`sub esp, 10h   ; reserve space for`
`               ; local variables`
…
`mov esp, ebp   ; load esp with ebp`
`pop ebp        ; restore ebp`
`ret            ; return`

- Typical *prolog* / *epilog*

Stack Frames

- Each function call has a so called *stack frame*
  - Input and output parameters (*stack parameters*)
  - Return address
  - Saved register contents (old `ebp`)
  - Local variables
- Base/frame pointer to address stack parameters and local variables (*FPO disabled*)



Source: http://upload.wikimedia.org/wikipedia/en/a/a7/ProgramCallStack2.png

Example

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```

Assembler is executed step by step in the next few slides

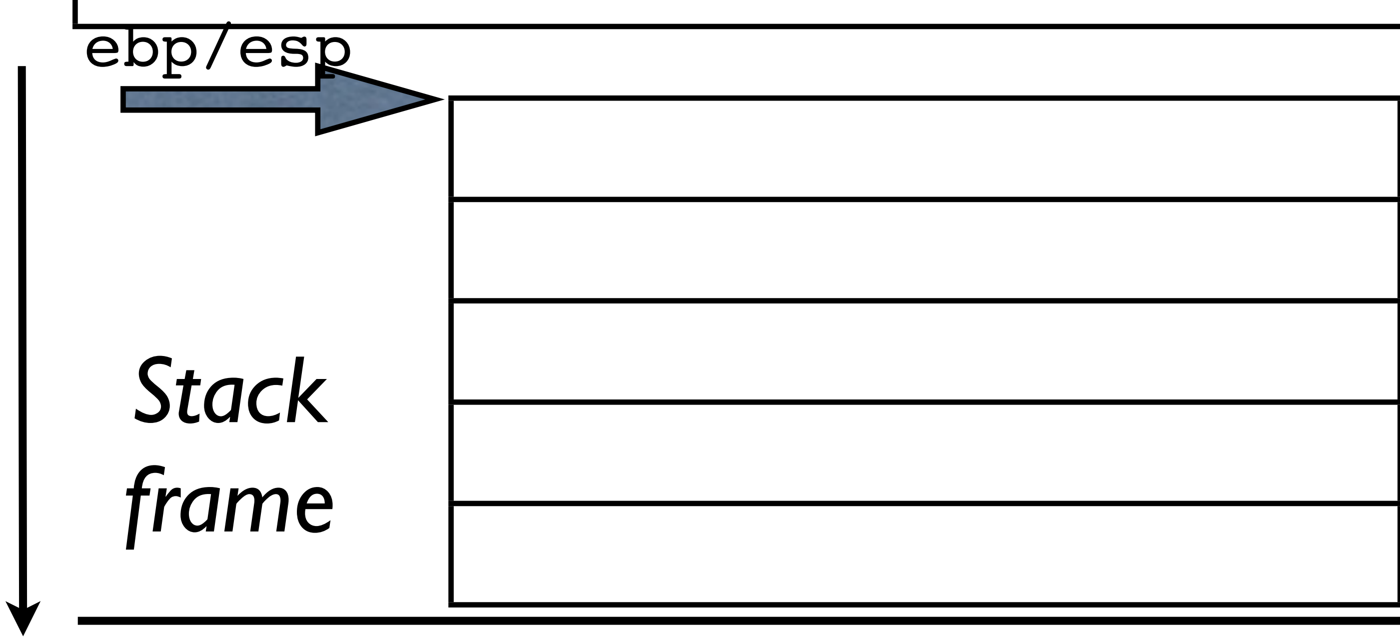
Assembler

```
   push 4
   push [var_y]
   call func1
   mov [var_x], eax   ;RET

func1:
   push ebp
   mov ebp, esp                Prolog
   sub esp, 4
   mov ecx, [ebp+0x8]
   mul ecx, [ebp+0xC]
   mov [ebp-4], ecx
   mov eax, [ebp-4]
   mov esp, ebp
   pop ebp                     Epilog
   ret 8
```

Example

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```



Assembler

Example

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```

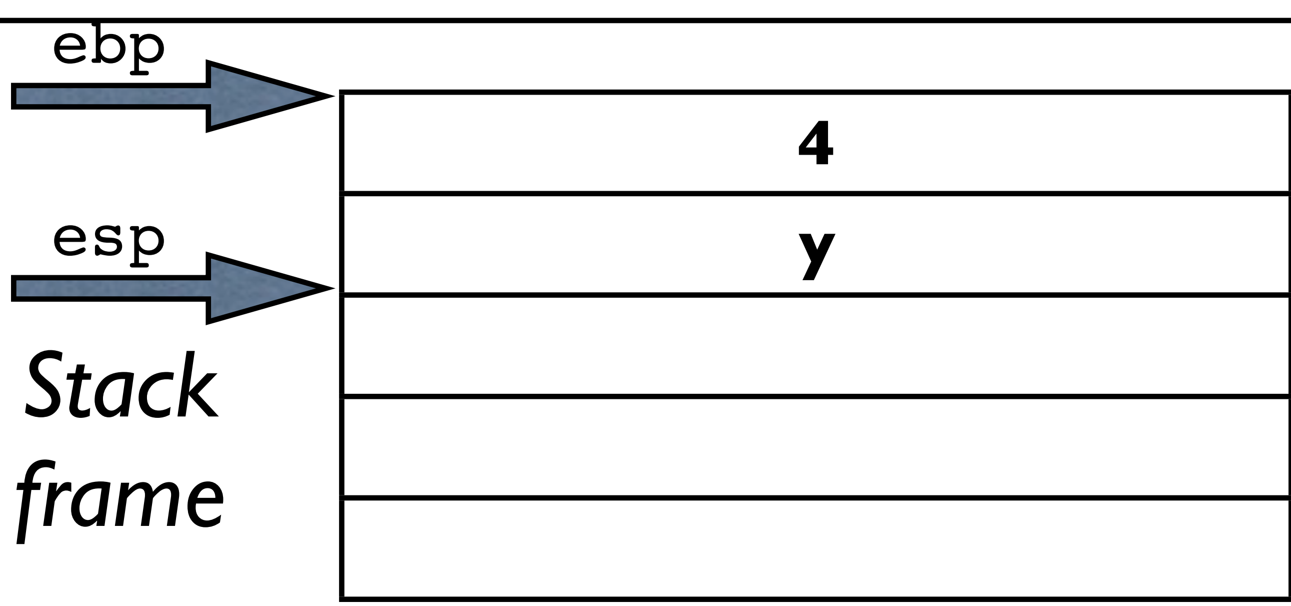
Assembler

```
push 4
```

Example

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```



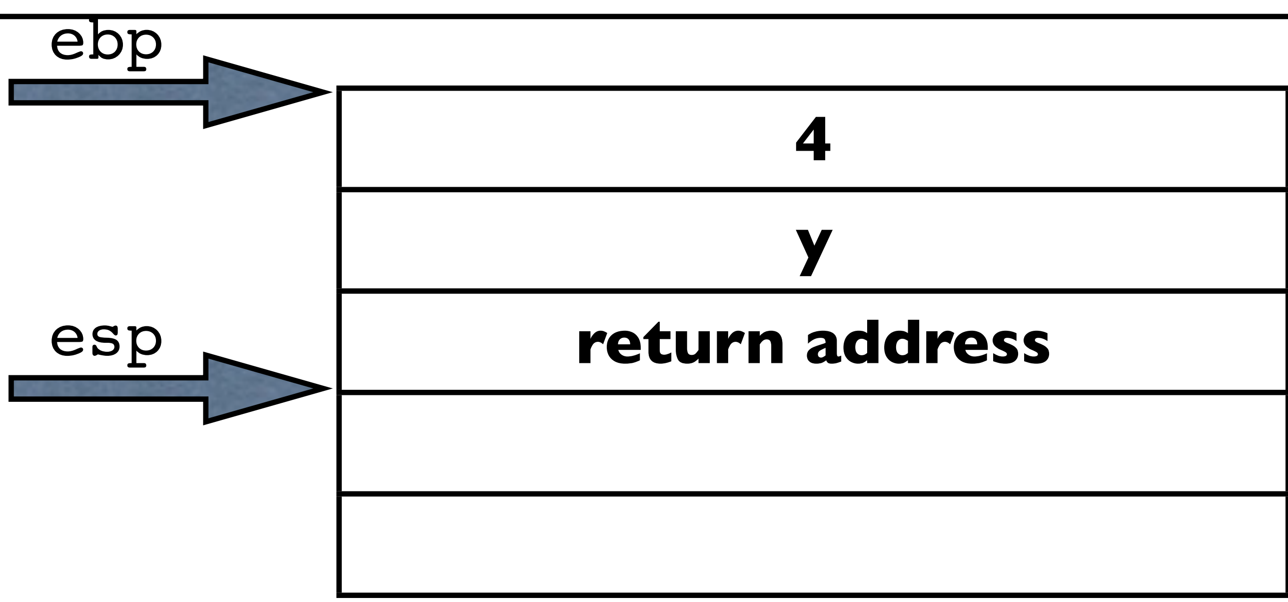
Assembler

```
push 4
push [var_y]
```

Example

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```



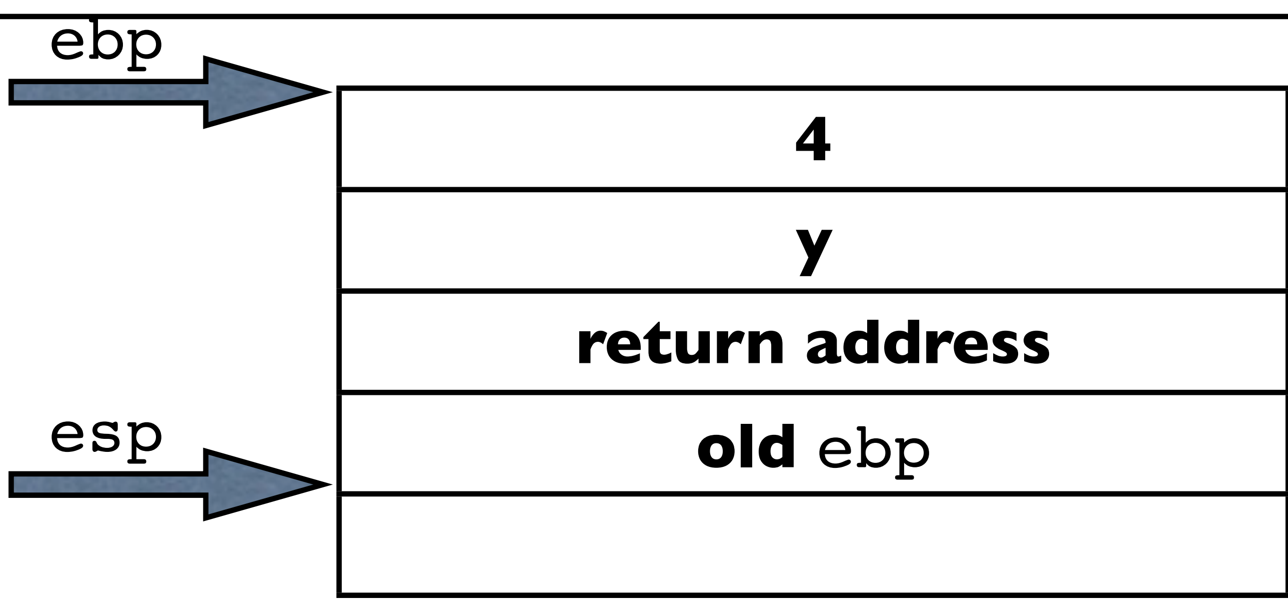
Assembler

```
push 4
push [var_y]
call func1
```

Example

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```



Assembler

```
   push 4
   push [var_y]
   call func1


func1:
   push ebp
```

Example

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```

Assembler

```
   push 4
   push [var_y]
   call func1


func1:
   push ebp
   mov ebp, esp
```

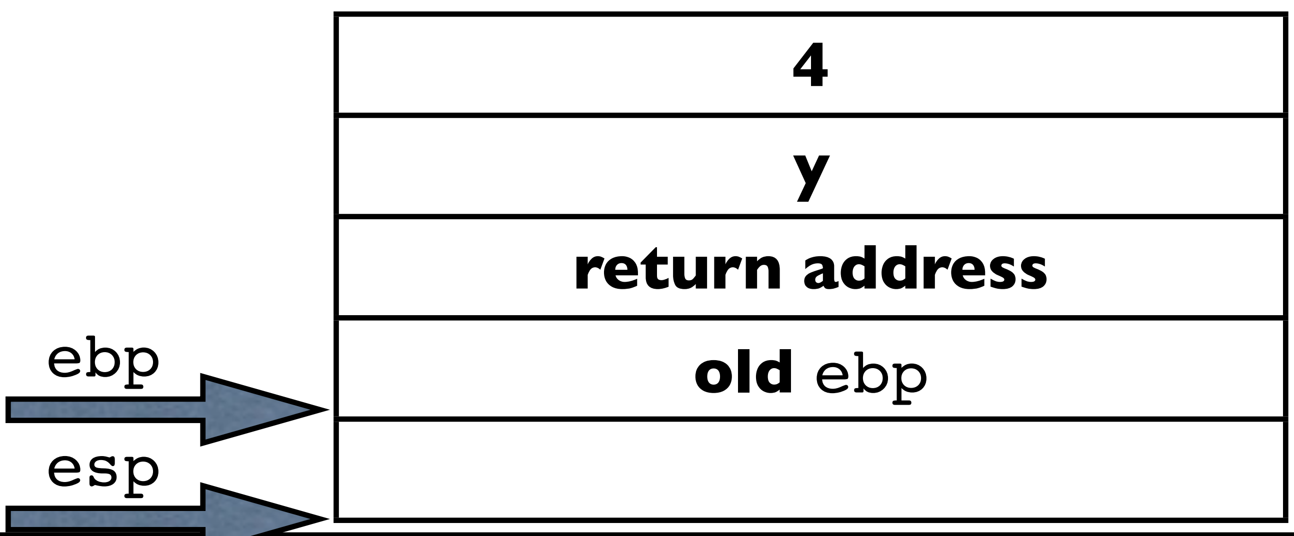
4
y
return address
old ebp

ebp/esp

Example

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```



Assembler

```
   push 4
   push [var_y]
   call func1


func1:
   push ebp
   mov ebp, esp
   sub esp, 4
```

Example

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```

Assembler

```
   push 4
   push [var_y]
   call func1


func1:
   push ebp
   mov ebp, esp
   sub esp, 4
   mov ecx, [ebp+0x8]
   mul ecx, [ebp+0xC]
   mov [ebp-4], ecx
```

	ebp + 0xC	**4**
	ebp + 0x8	**y**
	ebp + 0x4	**return addr.**
ebp →	ebp	**old** ebp
esp →	ebp - 0x4	**z = y * 4**

Example

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```

Assembler

```
   push 4
   push [var_y]
   call func1


func1:
   push ebp
   mov ebp, esp
   sub esp, 4
   mov ecx, [ebp+0x8]
   mul ecx, [ebp+0xC]
   mov [ebp-4], ecx
```

	ebp + 0xC	**4**
	ebp + 0x8	**y**
	ebp + 0x4	**return addr.**
ebp →	ebp	**old** ebp
esp →	ebp - 0x4	**z = y * 4**

First stack argument can be found at [ebp+08], with local variables typically at a negative displacement from ebp.

Example

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```

ebp + 0xC	**4**
ebp + 0x8	**y**
ebp + 0x4	**return addr.**
ebp	**old** ebp
ebp - 0x4	**z = y * 4**

ebp

esp

Assembler

```
   push 4
   push [var_y]
   call func1


func1:
   push ebp
   mov ebp, esp
   sub esp, 4
   mov ecx, [ebp+0x8]
   mul ecx, [ebp+0xC]
   mov [ebp-4], ecx
   mov eax, [ebp-4]
```

Example

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```

4
y
return address
old ebp

ebp/esp

Assembler

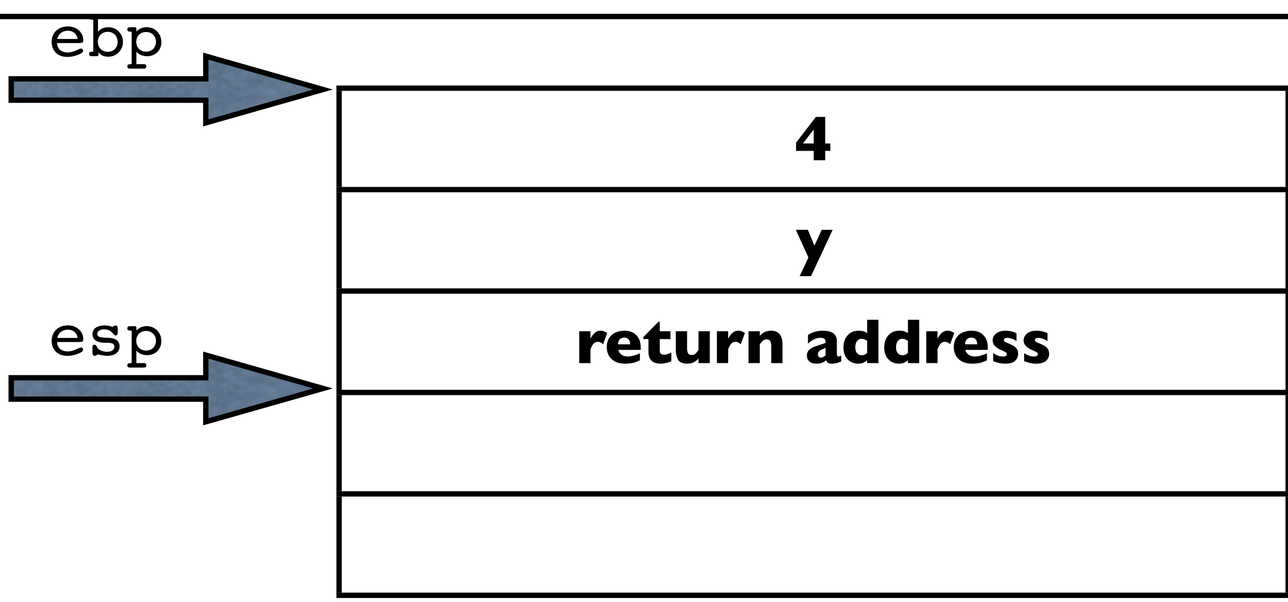
```
   push 4
   push [var_y]
   call func1


func1:
   push ebp
   mov ebp, esp
   sub esp, 4
   mov ecx, [ebp+0x8]
   mul ecx, [ebp+0xC]
   mov [ebp-4], ecx
   mov eax, [ebp-4]
   mov esp, ebp
```

Example

Systems Security
Ruhr-University Bochum

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```



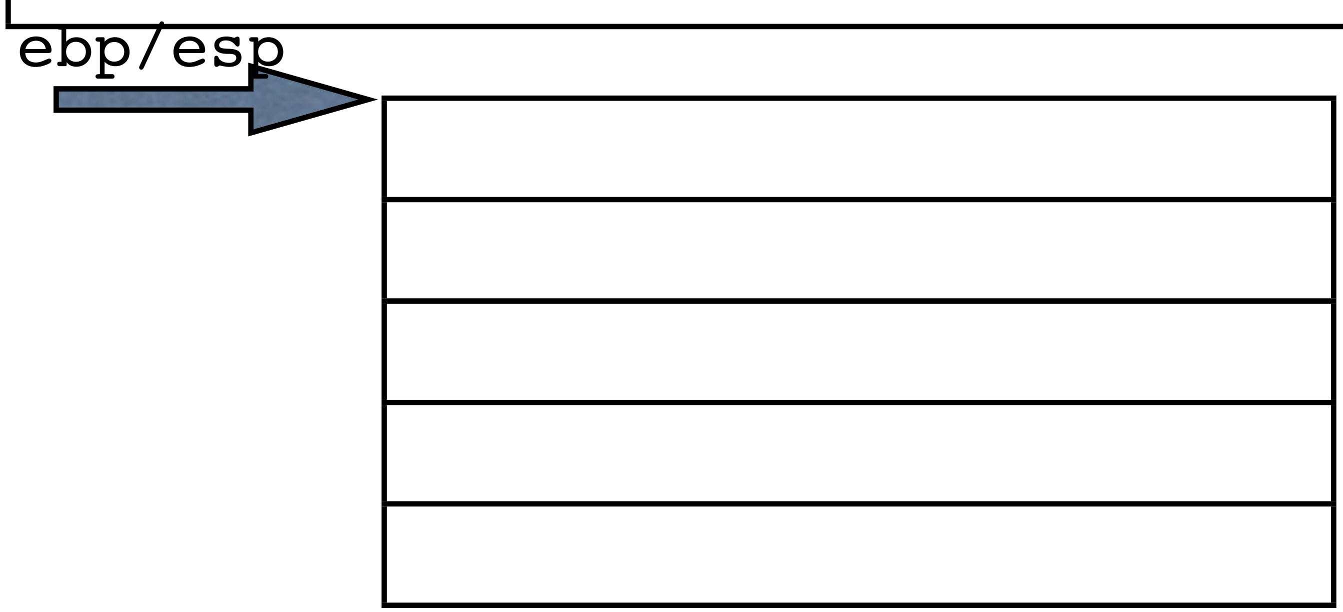
Assembler

```
   push 4
   push [var_y]
   call func1


func1:
   push ebp
   mov ebp, esp
   sub esp, 4
   mov ecx, [ebp+0x8]
   mul ecx, [ebp+0xC]
   mov [ebp-4], ecx
   mov eax, [ebp-4]
   mov esp, ebp
   pop ebp
```

Example

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```



Assembler

```
   push 4
   push [var_y]
   call func1


func1:
   push ebp
   mov ebp, esp
   sub esp, 4
   mov ecx, [ebp+0x8]
   mul ecx, [ebp+0xC]
   mov [ebp-4], ecx
   mov eax, [ebp-4]
   mov esp, ebp
   pop ebp
   ret 8
```

Example

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```

ebp + 0xC	**4**
ebp + 0x8	**y**
ebp + 0x4	**return addr.**
ebp	**old** ebp
ebp - 0x4	**z**

ebp

esp

Assembler

```
   push 4
   push [var_y]
   call func1
   mov [var_x], eax   ;RET

func1:
   push ebp
   mov ebp, esp
   sub esp, 4
   mov ecx, [ebp+0x8]
   mul ecx, [ebp+0xC]
   mov [ebp-4], ecx
   mov eax, [ebp-4]
   mov esp, ebp
   pop ebp
   ret 8
```

Example II

High-level language

```
void f(int param1, int param2) {
   int var1 = 0;
   int var2 = 1;
   param1 = var1;
   param2 = var2;
}
```

Assembler

```
push    ebp
mov     ebp, esp
sub     esp, 8
mov     [ebp-4], 0
mov     [ebp-8], 1
mov     eax, [ebp-4]
mov     [ebp+8], eax
mov     ecx, [ebp-8]
mov     [ebp+12], ecx
mov     esp, ebp
pop     ebp
retn
```

Stack frame

ebp + 0xC	**param2**
ebp + 0x8	**param1**
ebp + 0x4	**return addr.**
ebp	**old** ebp
ebp - 0x4	**var1**
ebp - 0x8	**var2**

ebp →
esp →

Calling Conventions

- Three problems
  - How are parameters passed?
  - When is the stack pointer modified during a call to a subroutine?
  - Which registers need to be saved during a subroutine call?
- No problem within the same program, compiler decides which *calling convention* to use

Calling Conventions

- What happens with external code?
  - For example: API in external library
- Calling and called function need to use same calling convention
- Needs to be known during compile time
- There are four relevant calling conventions: *cdecl*, *stdcall*, *fastcall* and *thiscall*

Calling Conventions

Systems Security
Ruhr-University Bochum

cdecl	stdcall	fastcall
- All parameters are pushed to stack - **Calling** function cleans the stack after the function call returns - C semantic - *Advantage*: variable number of parameters (printf & co.)	- All parameters are pushed to stack - **Callee** is responsible to cleanup the stack (e.g., `ret 12`)	- First three parameters in `eax`, `edx` and `ecx` (Windows: first two parameters in `ecx` and `edx`) - All other parameters on stack - **Callee** is responsible to cleanup that stack

Calling Conventions

cdecl	stdcall	fastcall
- eax, ecx, and edx are available for use in the function - Return value available in eax	- eax, ecx, and edx are available for use in the function - Return value available in eax - Standard calling convention for Win32 API	- Return value available in eax - Typically only used internally by OS

Example

cdecl	stdcall	fastcall

Call:
`push 3`
`push 2`
`push 1`
`call F`
`add  esp, 12`
`cmp  eax, 0`
`jz   error`
 	
Subroutine F:
`push ebp`
`mov  ebp, esp`
...
`pop ebp`
`ret`
	Call:
	`push 3`
	`push 2`
	`push 1`
	`call F`
	`cmp  eax, 0`
	`jz   error`
	
	Subroutine F:
	`push ebp`
	`mov  ebp, esp`
	...
	`pop ebp`
	`ret 12`
		Call:
		`mov  ecx, 3`
		`mov  edx, 2`
		`mov  eax, 1`
		`call F`
		`cmp  eax, 0`
		`jz   error`
		
		Subroutine F:
		`push ebp`
		`mov  ebp, esp`
		...
		`pop ebp`
		`ret`

Example

- Content of registers need to be saved when entering the function...
- ... and restored when leaving the function again
- First push to stack, then pop in reverse order

```
push ebx
push esi
push ebp
mov ebp, esp
...
mov ebx, 0
...
mov esi, DEADBEEFh
...
pop ebp
pop esi
pop ebx
ret
```

cdecl

- Advantage of cdecl if number of parameter is not fixed
  - Called subroutine does not need to know the number of parameters
  - Supports semantic required by C

printf(„Integer: %d“, i, someothervar);

```
push [someothervar]
push [i]
push offset IntegerStr
call printf
add  esp, 12
ret
```

Summary: Calling function

- Push parameter to stack (or pass in registers)
- Return value in `eax`
- Optionally: remove parameters from stack

Summary: Callee

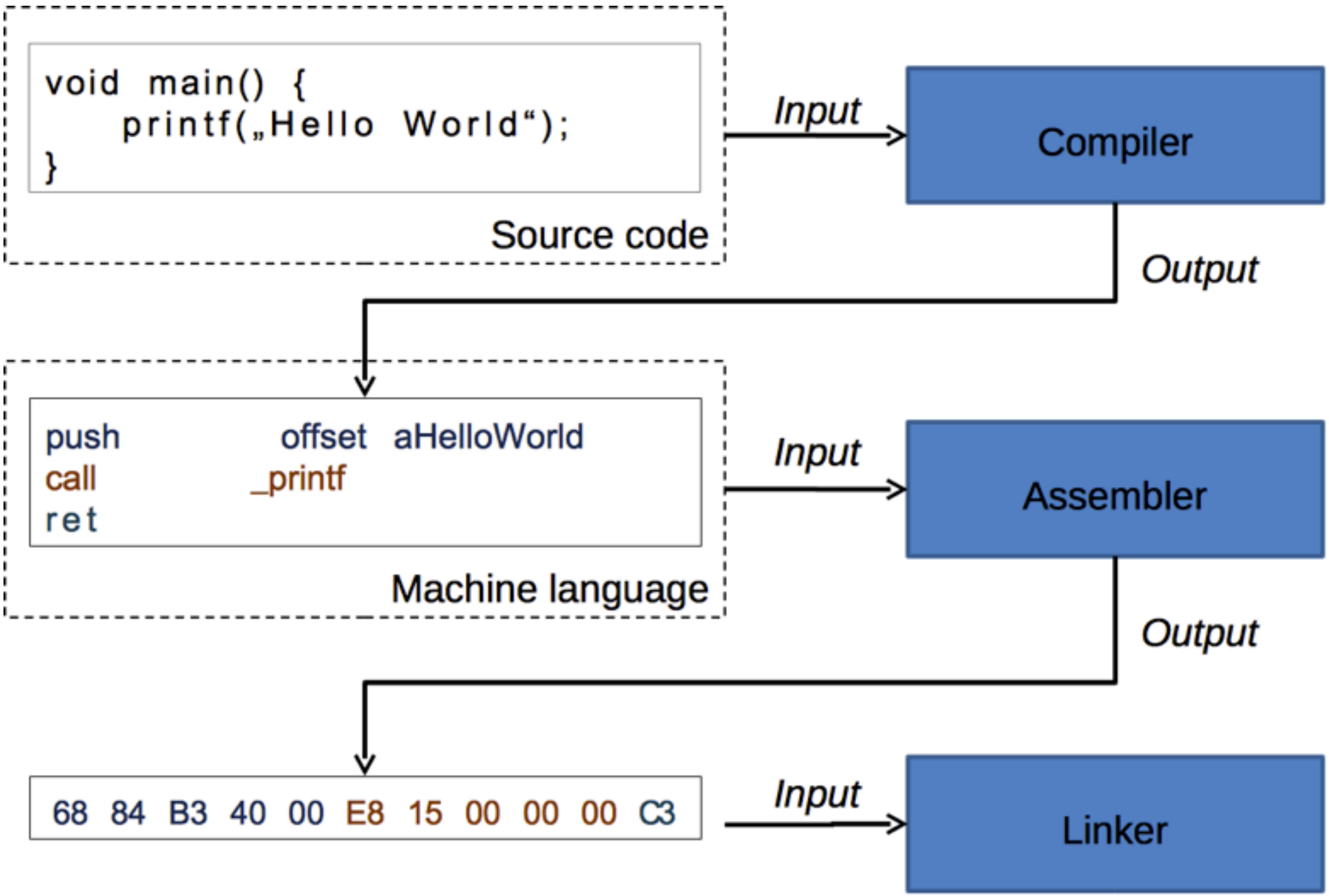
- Prolog
  - Save old `ebp`, `esp` is new `ebp`
  - Reserve memory space for local variables on stack
- Epilog
  - Restore old `ebp`
  - Jump to return address
  - Optionally: remove parameters from stack

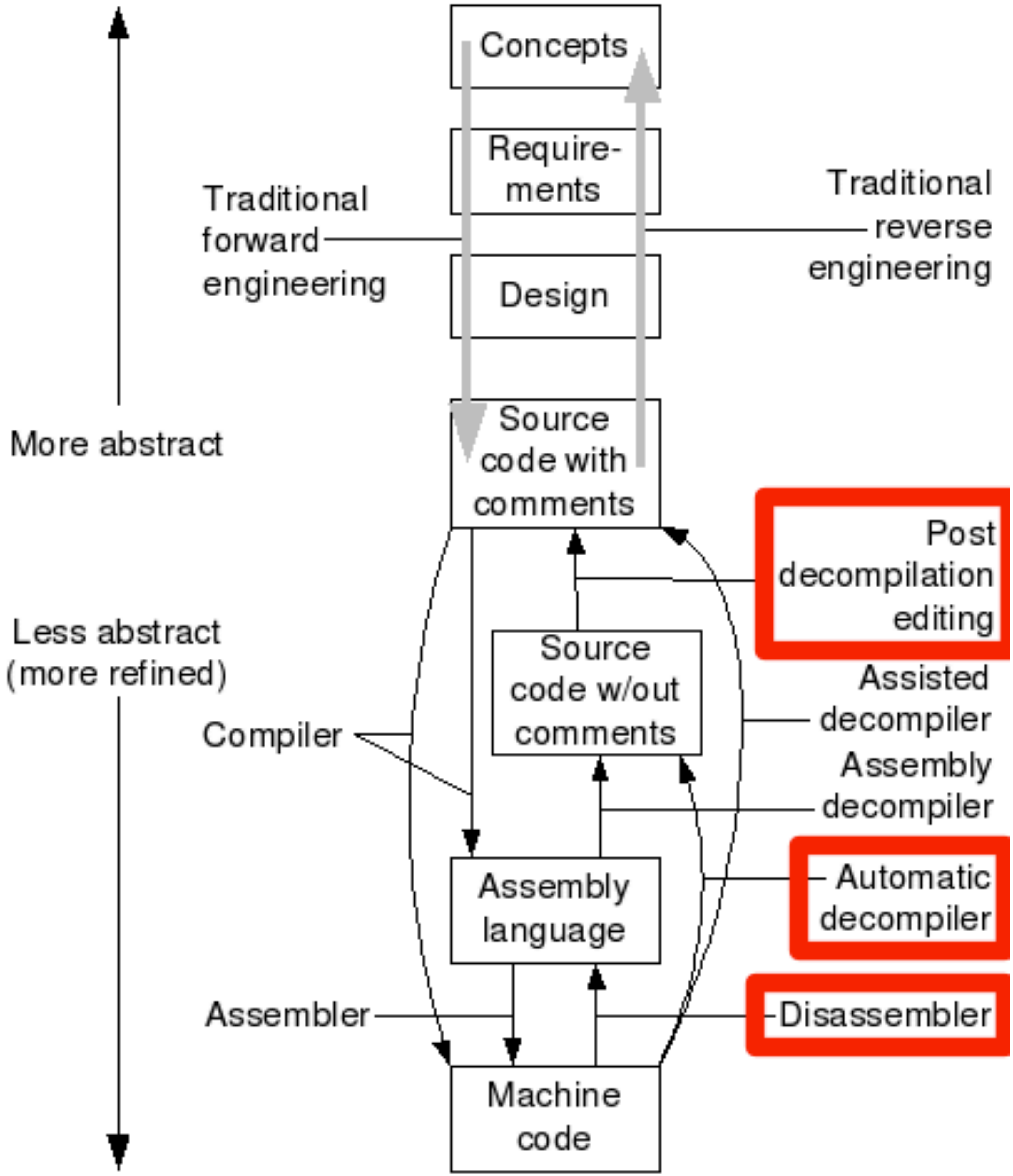
Higher-level Structures

Higher-level Languages

- Programming in assembler is cumbersome
  - Hard to understand and maintain
  - Code is not portable
  - Nowadays commonly used in performance-critical applications, malware/shellcode, and similar areas
- Typically programmers implement in higher-level languages like C/C++, Java, Python, ...

Compiler Overview





Source: http://www.program-transformation.org/Transform/DecompilationAndReverseEngineering

Reverse Engineering

- Typical goal: reconstruct code information and data structures from a given binary program
  - Assembler ⇒ Higher-level language
  - Several exercises deal with this topic
- To understand this, we take a look at examples
  - Higher-level language ⇒ Assembler
  - Getting a feeling of how compilers transform code

Example

High-level language

```
x = func1(y, 4);

stdcall int func1(int a, int b )
{
   int z;
   z = a * b;
   return z;
}
```

Assembler

```
   push 4
   push [var_y]
   call func1
   mov [var_x], eax   ;RET

func1:
   push ebp
   mov ebp, esp
   sub esp, 4
   mov ecx, [ebp+0x8]
   mul ecx, [ebp+0xC]
   mov [ebp-4], ecx
   mov eax, [ebp-4]
   mov esp, ebp
   pop ebp
   ret 8
```

Stack frame

ebp + 0xC	**4**
ebp + 0x8	**y**
ebp + 0x4	**return addr.**
ebp	**old** ebp
ebp - 0x4	**z**

ebp →
esp →

Control-flow Structures

- Conditional expressions
  - IF ... THEN ...
- Branches
  - IF ... THEN ... ELSE ...
  - SWITCH ... CASE ...
- Loops (WHILE ... DO, DO ... WHILE, FOR ... DO ...)
- Function calls

Conditional Expressions

High-level language

```
IF( SomeVar == 0 ) THEN
    function1()
```

Assembler

```
   mov eax, [SomeVar]
   test eax, eax
   jnz lab_end
   call function1
lab_end:
   ...
```

Branches

High-level language

```
IF( Var1 == 1 )
   THEN function1()
   ELSE IF( Var1 == 2 )
      THEN function2()
      ELSE function3()
```

Assembler

```
   cmp [Var1],1
   jne lab_else1
   call function1
   jmp lab_end
lab_else1:
   cmp [Var1],2
   jne lab_else2
   call function2
   jmp lab_end
lab_else2:
   call function3
lab_end:
   ...
```

Expressions: OR

High-level language

```
IF( Var1 == 1 ||
    Var1 == 2 ||
    Var2 >= 5 )
  THEN function1()
```

Assembler

```
   cmp [Var1], 1
   je lab_then
   cmp [Var1], 2
   je lab_then
   cmp [Var2], 5
   jae lab_then
   jmp lab_end
lab_then:
   call function1
lab_end:
   …
```

Switch Statements

- Switch statements are common

```
switch( condition ) {
  case value1: …; break;
  case value2: …; break;
  case value3: …; break;
  default: …; break }
```

- Often implemented as
    - *Table*: if values are near each other, complexity O(1)
    - *Tree*: if values are far away, complexity O(log(n))

Example: Table Lookup I

High-level language

```
switch( Var1 )
{
   case 0: RC = 0; break;
   case 1: RC = 1; break;
   case 2: RC = 2; break;
   case 3: RC = 3; break;
   default: RC = 0; break;
}
```

```
switch_table:
0x00: dd offset lab_case0
0x04: dd offset lab_case1
0x08: dd offset lab_case2
0x0C: dd offset lab_case3
```

Assembler

```
   cmp [Var1], 3
   ja lab_default
   mov ecx, [Var1]
   jmp switch_table[ecx*4]
lab_case0:
   mov [RC], 0
   jmp lab_end
lab_case1:
   mov [RC], 1
   jmp lab_end
   …
lab_default
   mov [RC], 0
   jmp lab_end
lab_end:
   …
```

Example: Table Lookup II

High-level language

```
switch( Var1 )
{
   case 0: RC = 0; break;
   case 1: RC = 1; break;
   case 2: RC = 2; break;
   case 4: RC = 4; break;
   default: RC = 0; break;
}
```

```
switch_table:
0x00: dd offset lab_case0
0x04: dd offset lab_case1
0x08: dd offset lab_case2
0x0C: dd offset lab_default
0x10: dd offset lab_case4
```

Assembler

```
   cmp [Var1], 4
   ja lab_default
   mov ecx, [Var1]
   jmp switch_table[ecx*4]
lab_case0:
   mov [RC], 0
   jmp lab_end
   …
lab_case4:
   mov [RC], 4
   jmp lab_end
   …
lab_default
   mov [RC], 0
   jmp lab_end
lab_end:
```

Example: Table Lookup II

High-level language

```
switch( Var1 )
{
   case 0: RC = 0; break;
   case 1: RC = 1; break;
   case 2: RC = 2; break;
   case 4: R
   default:
}
```

```
switch_table:
0x00: dd offset lab_case0
0x04: dd offset lab_case1
0x08: dd offset lab_case2
0x0C: dd offset lab_default
0x10: dd offset lab_case4
```

Assembler

```
   cmp [Var1], 4
   ja lab_default
   mov ecx, [Var1]
   jmp switch_table[ecx*4]
lab_case0:
   mov [RC], 4
   jmp lab_end
   …
lab_default
   mov [RC], 0
   jmp lab_end
lab_end:
```

If there are "holes": use default value in lookup table

Example: Tree Lookup

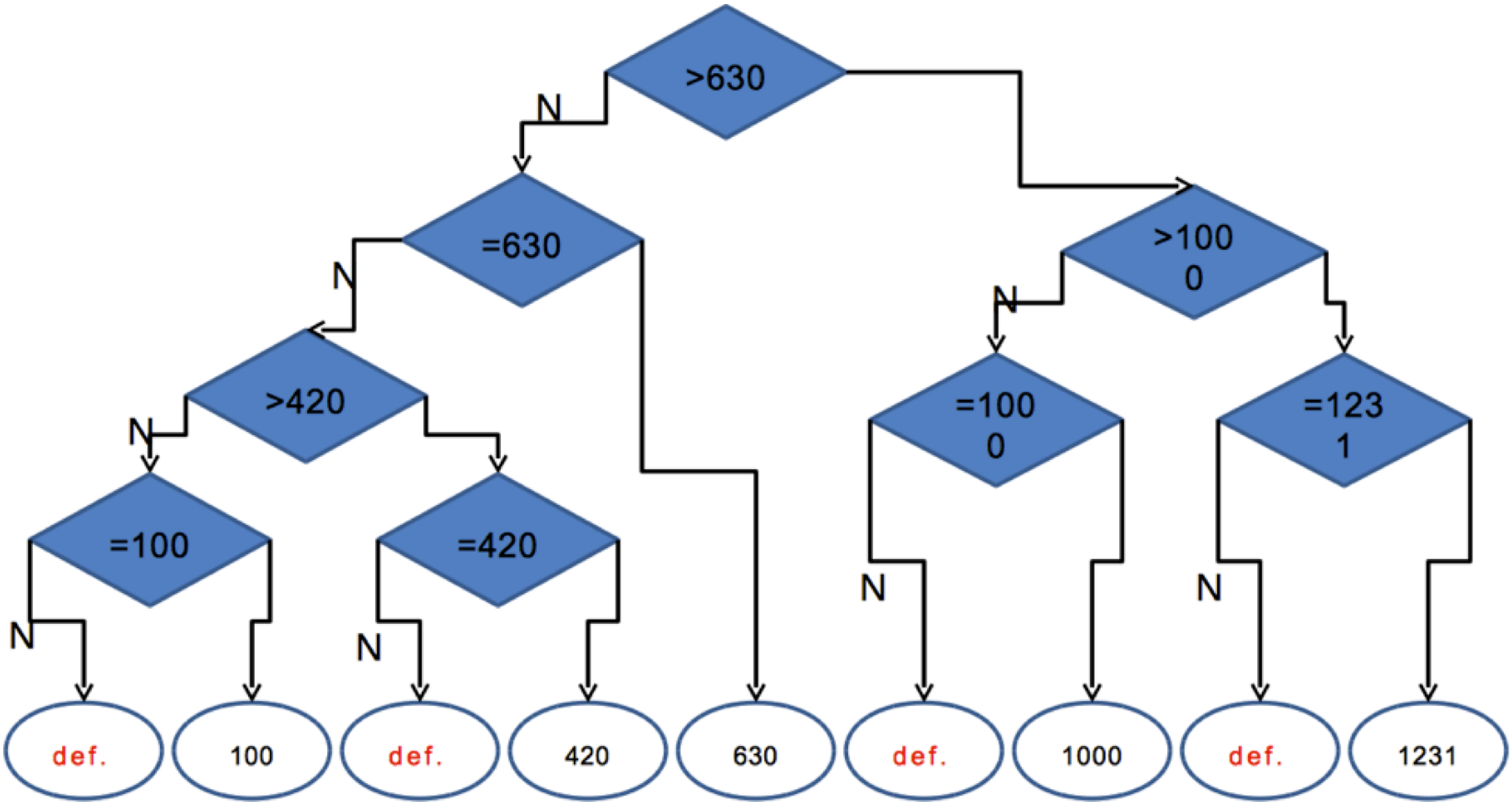
High-level language

```
switch( Var1 )
{
   case 0x0: …
   case 0x100: …
   case 0x150: …
   case 0x172: …
   case 0x210: …
   case 0x230:  …
   case 0x2A0:  …
   default: …
}
```

Assembler

```
   cmp [Var1], 0x172
   jg lab_above_0x172
   cmp [Var1], 0x172
   je lab_0x172
   cmp [Var1], 0x100
   jg lab_above_0x100
   cmp [Var1], 0x100
   je lab_0x100
   cmp [Var1], 0x0
   je lab_0x0
   jmp lab_default
lab_above_0x100:
   cmp [Var1], 0x150
   je lab_0x150
   jmp lab_default
lab_above_0172:   …
```

Example: Tree Lookup



Questions?

Contact:
Prof. Thorsten Holz
thorsten.holz@rub.de
@thorstenholz on Twitter

More information:
http://syssec.rub.de
http://moodle.rub.de

Sources

- Lecture *Software Reverse Engineering* at University of Mannheim, spring term 2010 (Ralf Hund, Carsten Willems and Felix Freiling)