Code: analysis, bugs, and security
supported by Bitdefender

Assembly Language

Marius Minea

marius@cs.upt.ro

11 October 2017

Computing is about abstraction

Many high-level languages with scores of features

Processor hardware: much simpler, less variation

Compiler must bridge this gap general rules/schemes of translation

adapted to the processor architecture

additionally: optimizations – make code short, fast

Warm-up quiz

What will the following print?

```c
int i = 5;
printf("%d %d %d\n", i++, i++, i++);
```

Warm-up quiz

What will the following print?

```
int i = 5;
printf("%d %d %d\n", i++, i++, i++);
```

Compiler warns: behavior is undefined
several *unsequenced* side-effects (increments) on same object (`i`)
(orderings between these side-effects and value computations unknown)

But typical compilers will produce code consistently

Towards assembly: three-address code

Expressions can be arbitrarily complex

Most often, we have simple expressions like:

`x = 2 * y` or `z = x + y`

two operands, one result

three-address code:

(at most) three addresses (variables, pointers) in instruction

manageable complexity for humans, algorithms

closer to processor capabilities

Can do source-to-source transformation of C to three-address code
e.g., with CIL analysis infrastructure (Necula et al., Berkeley)

Why registers

Would three-address code be enough?

Processor instructions could have an opcode (for basic operations) + (at most) three addresses.

You might do this for a virtual machine in a compiler class project
Why not in a real processor?

Why registers

Would three-address code be enough?

Processor instructions could have an opcode (for basic operations) + (at most) three addresses.

You might do this for a virtual machine in a compiler class project
Why not in a real processor?

speed: memory access is slow

code *size*: three arbitrary addresses make a long instruction

Instruction set design

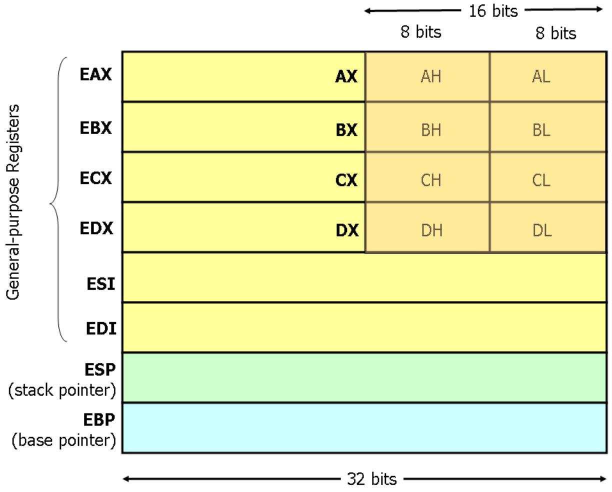
From early simple processors to CISC to RISC

Complex Instruction Set Computer
powerful instructions, complex addressing modes
multi-step operations in same instruction

Reduced Instruction Set Computer
uniform short instructions (one word)
general-purpose registers (with same role)
simple addressing, load/store architecture
(separate memory access and arithmetic)

ultimately, seems microarchitecture has more performance impact

x86 Architecture: Registers



On 64-bit architectures: extended registers: rax, rbx, etc.

http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Opcodes and operands

Instructions have 1 to several bytes

Opcode: says what instruction does
  1-byte instructions: `ret`, `push`/`pop`/`inc`/`dec` *reg*

Operands:
  register (8, 16 or 32-bit) + 64
  constant (8, 16 or 32-bit) + 64
  (contents of) memory address

Size directives

For a memory access, must indicate amount of data read/written
  1, 2, 4, 8 bytes

In C, given by the type of the pointer: `char *`, `int16_t *`, etc.

In assembly, must specify explicitly ⇒ different instructions

```
mov BYTE PTR [ebx], 7
mov WORD PTR [ebx], 7
mov DWORD PTR [ebx], 7
```

Stack: push and pop

Simplest memory transfer instructions

push: first decrement, then put value on stack

```
sp -= 4        // for 32-bit arch
[sp] = value
```

pop: take value from stack pointer, then increment

```
value = [sp]
sp += 4       // for 32-bit arch
```

Memory transfer (data movement) instructions

```
mov eax, [ebx]   ; [ ] = contents of address w/ given valu
mov [var], ebx   ; var = 32-bit const addr
mov eax, [esi-4]
mov [esi+eax], cl ; register cl = one byte
mov edx, [esi+4*ebx]
```

Examples: `http://www.cs.virginia.edu/~evans/cs216/guides/x86.html`

Basic Arithmetic & Bitwise ops

Arithmetic:

`add, sub, mul, div`
`imul`, `idiv` (signed)
`inc`, `dec`

Logical (bitwise): `and`, `or`, `xor`, `not`

`xor bx, bx` // short way to zero a register

Shifting: by constant bitcount, or value or reg `cl`

```
shl [mem], 3
shr dx, cl
```

All of these affect *flags*

The Flags Register

CF (carry): set when *unsigned* result does not fit

OF (overflow): set when *signed* result does not fit

SF (sign): arithmetic / logic result is negative

ZF (zerox): arithmetic / logic result is zero

AC (auxiliary carry): from bit 3 to bit 4 in 8-bit operand

PF (parity): of low-order byte: 1 if even number of 1 bits

Calling conventions

When calling a function, several options to consider:

Where to pass arguments ? (stack or registers?)

Argument passing order (from left or from right)?

Who cleans up the stack? (caller or callee)

Who saves registers? (caller or callee)

cdecl convention

32-bit x86, many compilers, Unix-like systems

args passed on stack, right to left (allows varargs)

result returned in `eax`

caller saves `eax`, `ecx`, `edx`, callee saves rest

caller cleans up stack

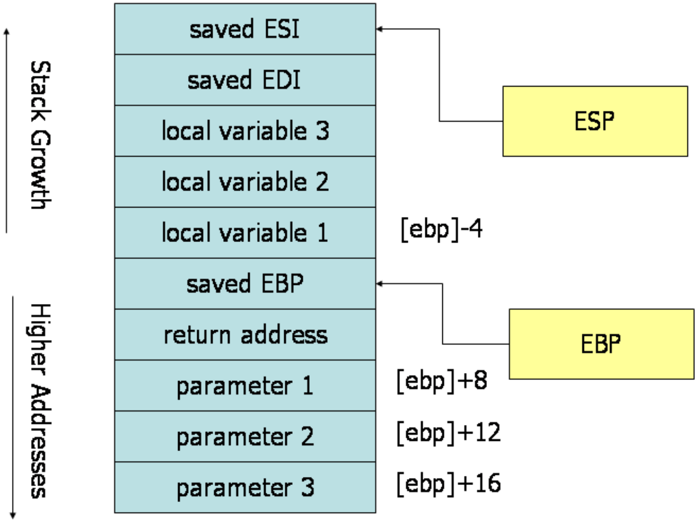
stack frame multiple of 16 bytes (since gcc 4.5)

Compile to assembly (`cc` could be `gcc`, `clang`, etc.)

```
cc -S -masm=intel file.c
```

Extra options: `-O2` to optimize

`-m32` compiles to 32 bits on 64-bit system



http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Typical function prologue/epilogue

```
push ebp      ; save old value of base pointer
mov ebp, esp ; set to current stack pointer
sub esp, size ; make room for local vars
...           ; function code
mov esp, ebp ; if sp not at initial value
pop ebp       ; restore to prev. stack frame
ret
```

optionally, high-level function enter/exit

```
enter size, 0          ; 0 for non-nested
```

like

```
push ebp
mov ebp, esp
sub esp, size
```

```
leave
```

like

```
mov esp, ebp
pop ebp
```

Other calling conventions

`stdcall`: Microsoft

- args also right to left, *callee* cleans up stack
  - cannot have variable-length arguments

`syscall`

- like `cdecl` but does not save AX, CX, DX

x86-64 calling conventions

64-bit arch has 8 more registers ⇒ can use to pass values

System V AMD64 ABI

first 6 args passed in `rdi, rsi, rdx, rcx, r8, r9`
return value in `rax` and `rdx`

Code example: variable-length arguments

How would you implement

```
int myprintf(const char *fmt, ...);
```

knowing that arguments are passed from right to left ?

Recognizing function prologues / epilogues

Important in reverse engineering

May not know all entry points

May not be able to follow all function calls
  e.g. indirect calls, through pointers in a table

Standard prologues/epilogues help disassembler detect functions

Control flow: jumps and calls

`jmp` *address*
`call` *address*

Should *address* be absolute or relative to the progam counter?

Control flow: jumps and calls

`jmp` *address*
`call` *address*

Should *address* be absolute or relative to the progam counter?

relative: important to have *relocatable code*
  (can load at any address in memory)

Absolute jump / call instructions also exist.

Comparison operators

`cmp` *op1*, *op2*

like (signed) subtraction, but does not change left operand

`test` *op1*, *op2*

like bitwise AND, but does not change left operand

both set flags ⇒use for *conditional jumps*

Conditional jumps

Based on a variety of flags (set by `cmp` / `test`)

JA (above)	CF = 0 and ZF = 0
JB (below)	CF = 1
JC (carry)	CF = 1 (same as JB)
JE (equal)	ZF = 1
JG (greater)	ZF = 0 and SF = OF
JL (less)	SF != OF
JO (overflow)	OF = 1
JS (sign)	SF = 1
JZ (zero)	ZF = 1

also negations (JNA, JNB, etc.) + nonstrict cmp (JLE, JGE, etc)
some mnemonics mean same thing: JNGE = JL

Make common case fast

conditional jumps have near versions with 8-bit offset

Indirect jumps. Jump tables

from compilation of `switch` statement

```
typedef enum { ADD, SUB, MUL, DIV, MOD, AND, OR, XOR } op_t;

int calc(int op, int a, int b) {
  switch(op) {
  case ADD: return a + b;
  case SUB: return a - b;
  case MUL: return a * b;
  case DIV: return a / b;
  case MOD: return a % b;
  case AND: return a & b;
  case OR: return a | b;
  case XOR: return a ^ b;
  }
  return 0;
}
```

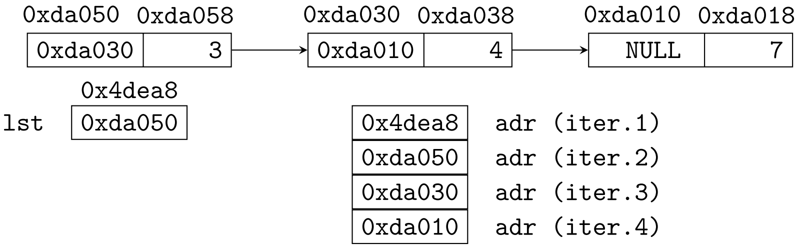
Indirect calls (pointer table)

```c
#include <stdio.h>
#include <stdlib.h>

typedef int (*intfn_t)(int, int);   // type of function pointer

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int mul(int a, int b) { return a * b; }
int idiv(int a, int b) { return a / b; }

intfn_t fntab[] = { add, sub, mul, idiv };

int main(int argc, char *argv[]) {
  if (argc != 2) return 1;
  unsigned op = atoi(argv[1]);
  if (op < 4) printf("%d\n", fntab[op](7, 3));
}
```

List iteration with pointers to pointers

```
typedef struct il { struct il *nxt; int el; } *intlist_t;

intlist_t ins_last(intlist_t lst, int val) {
  intlist_t *adr;     // pointer to cell pointer
  for (adr = &lst; *adr; adr = &(*adr)->nxt);
  // now adr is address of nxt field in last cell
  if ((*adr = malloc(sizeof(struct il)))) {
    (*adr)->el = val; (*adr)->nxt = NULL;
  }
  return lst;
}
```

In picture, top row denotes *addresses* of individual fields


0xda050
0xda058
0xda030
3
0xda030
0xda038
0xda010
4
0xda010
0xda018
NULL
7
0x4dea8
lst
0xda050
0x4dea8
adr (iter.1)
0xda050
adr (iter.2)
0xda030
adr (iter.3)
0xda010
adr (iter.4)