System Programming

Assembly

H. Turgut Uyar Şima Uyar

2001-2014

Topics

x86 Processors

- Intel family of processors: x86 (32 bit), x64 (64 bit)
- very similar from the programming standpoint
- 8086: 16 bit processor, real mode
- 80386: 32 bit processor, protected mode (virtual memory)

Segments

- programs are divided into segments
- code segment: instructions
- data segment: initialized data
- bss segment: uninitialized data
- stack segment

8086 Registers

- 4 general purpose data registers
- 2 index registers
- 2 pointer registers
- 4 segment registers
- 2 control registers

Data Registers

- AX: accumulator register
- BX: base register
  - used to address data in memory
- CX: counter register
  - used as repetition counter in loop operations
- DX: data register
  - used in multiplication and division operations
- high and low halves can be accessed as 8-bit registers: AH-AL, BH-BL, CH-CL, DH-DL

Index and Pointer Registers

- index registers:
  - DI: data index
  - SI: stack index
  - they can be used like general purpose registers
- pointer registers:
  - BP: base pointer
  - SP: stack pointer

Segment Registers

- CS: code segment register
- DS: data segment register
- SS: stack segment register
- ES: extra segment register

Control Registers

- IP: instruction pointer
  - CS + IP: address of next instruction
- FLAGS: status conditions
  - ZF (zero), OF (overflow), SF (sign), CF (carry), PF (parity)

80386

- in 80386, registers are extended to 32 bits:
  EAX EBX ECX EDX ESI EDI EBP ESP
  EIP
- AX, BX, ..., BP, SP are still valid (lower 16 bits)
- AH, AL, ..., DH, DL are still valid

Operand Types

- register
- memory: offset from beginning of segment
- immediate: listed in the instruction itself
- implied: not explicitly specified

Basic Instructions

`mov dest, src`	move src to dest
`add dest, src`	add src to dest
`adc dest, src`	add src to dest with carry
`sub dest, src`	subtract src from dest
`sbb dest, src`	subtract src from dest with borrow
`inc dest`	increment dest
`dec dest`	decrement dest
`mul src`	multiply eax with src, result in edx:eax
`div src`	divide edx:eax by src, result in eax and edx

Bitwise Instructions

`not dest`	bitwise not (one's complement)
`and dest, src`	bitwise and
`or dest, src`	bitwise or
`xor dest, src`	bitwise xor
`neg dest`	negate (two's complement)
`shl dest, amount`	logical shift left
`shr dest, amount`	logical shift right
`asl dest, amount`	arithmetic shift left
`asr dest, amount`	arithmetic shift right
`rol dest, amount`	rotate left
`ror dest, amount`	rotate right
`rcl dest, amount`	rotate left with carry
`rcr dest, amount`	rotate right with carry

Branching Instructions

`jmp`	unconditional
`jz`	if ZF is set
`jnz`	if ZF is unset
`jo`	if OF is set
`jno`	if OF is unset
`js`	if SF is set
`jns`	if SF is unset
`jc`	if CF is set
`jnc`	if CF is unset
`jp`	if PF is set
`jnp`	if PF is unset

Branching Instructions

- `cmp vleft, vright`: compare vleft and vright

condition	signed	unsigned
vleft $=$ vright	`je`	`je`
vleft $\neq$ vright	`jne`	`jne`
vleft $<$ vright	`jl`	`jb`
vleft $\not<$ vright	`jnl`	`jnb`
vleft $\leq$ vright	`jle`	`jbe`
vleft $\not\leq$ vright	`jnle`	`jnbe`
vleft $>$ vright	`jg`	`ja`
vleft $\not>$ vright	`jng`	`jna`
vleft $\geq$ vright	`jge`	`jae`
vleft $\not\geq$ vright	`jnge`	`jnae`

Directives

- assembler needs extra info: *directives*
- not part of the instruction set
- labels: mark points in code and data
  - entry labels have to marked `global`
- segments
- data definition
- named constants: `equ`
  - no memory allocated

Code Template

```
segment .data
; initialized data definitions

segment .bss
; uninitialized data definitions

segment .text
global _start

_start:
    ; entry point
```


Data Definition

type	initialized	uninitialized
byte	`db`	`resb`
word	`dw`	`resw`
dword	`dd`	`resd`
qword	`dq`	`resq`
tword	`dt`	`rest`

Data Definition Examples

```
L1 db    0
L2 dw    1000
L3 dd    1A92h
L4 db    0, 1, 2, 3
L5 db    "w", "o", "r", "d", 0
L6 db    'word', 0
L7 times 100 db 0
L8 resb  1
L9 resw  100
```


Dereferencing

- plain label:
  address of memory

Example

```
mov eax, L1
```

- label in brackets:
  contents of memory

Example

```
mov eax, [L1]
```

System Calls

- system calls are implemented using software interrupt 80h

system call setup

eax ← system call number
ebx ← first argument
ecx ← second argument
edx ← third argument
int 80h

System Call Examples

- exit system call number: 1
- arg. 1: return status
  - 0: success, 1: failure

- read system call number: 3
- arg. 1: input descriptor
  - 0: stdin, 1: stdout, 2: stderr
- arg. 2: start of input buffer
- arg. 3: length of input

- write system call number: 4
- arg. 1: output descriptor
  - 0: stdin, 1: stdout, 2: stderr
- arg. 2: start of output buffer
- arg. 3: length of output

Example: Hello, world!

```
segment .data
msg db  "Hello, world!", 10
len equ 14
```

```
segment .text
global _start

_start:
    mov  eax, 4
    mov  ebx, 1
    mov  ecx, msg
    mov  edx, len
    int  80h

    mov  eax, 1
    mov  ebx, 0
    int  80h
```


References

Required Reading: Carter

- Chapter 1: Introduction
  - 1.2. Computer Organization
  - 1.3. Assembly Language

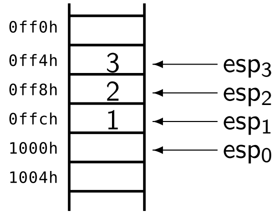
Stack

- the stack is accessed in 4-byte units

`push operand`
- subtract 4 from esp
- store operand to address [esp]

`pop register`
- store operand at address [esp] to register
- add 4 to esp

Stack Example




```
push dword 1
push dword 2
push dword 3
pop eax
pop ebx
pop ecx
```


Subroutine Call

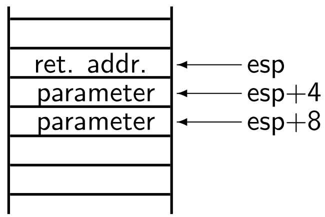
`call target`
- push address of next instruction
- jump to target

`ret`
- pop return address
- jump to return address

Stack Parameters

- called subroutine does not pop parameters
- accesses parameters on the stack

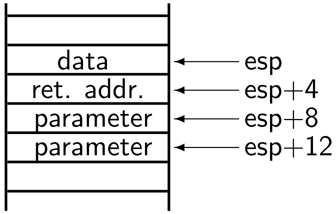
stack layout



Accessing Parameters

- offsets from esp may change

example: after a push



Accessing Parameters

- use ebp (frame pointer)

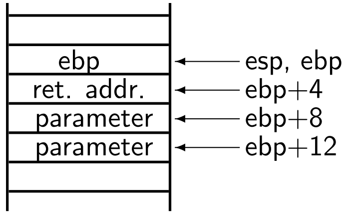
subroutine template

```
push ebp
mov  ebp, esp

...

pop  ebp
ret
```

stack layout



Example: Factorial

```
segment .bss
f    resd 1

segment .text

fact:
    push ebp
    mov  ebp, esp

    mov  dword [f], 1
    mov  ecx, [ebp+8]
```

```
back:
    mov  eax, [f]
    mul  ecx
    mov  [f], eax
    dec  ecx
    cmp  ecx, 1
    jne  back

    pop  ebp
    ret
```


Example: Calling Factorial

```
segment .data
k    dd    5

segment .bss
f    resd 1

segment .text
global _start

fact:
    ...
```

```
_start:
    push ebp
    mov  ebp, esp

    push dword [k]
    call fact
    add  esp, 4

    pop  ebp
    ret
```

Calling Conventions

- how will parameters be passed?
- if using stack:
  - in what order will the parameters be pushed?
  - who will remove parameters from the stack?
- how will the result be returned?
- which registers should remain unchanged?

C Calling Conventions

- parameters are passed via the stack
  - caller pushes parameters in reverse order
  - caller removes parameters from the stack
- result is returned over eax
- ebx, esi, edi, ebp, cs, ds, ss, es should remain unchanged

Calling C from Assembly

- to call a C function from Assembly:
- declare function as `extern`
- push arguments in reverse order
- call function
- adjust esp

Example: printf

```
segment .data
k    dd   5
intf db   "%d", 10, 0

segment .bss
f    resd 1

segment .text
global main
extern printf

fact:
    ...
```

```
main:
    ...

    push dword [k]
    call fact
    add  esp, 4

    push dword [f]
    push intf
    call printf
    add  esp, 8

    ...
```

C Variables

- global: in fixed memory locations
- static: same as global, only scope is different
- automatic: on stack
- register: in a register (if possible)
- volatile: do not optimize

Automatic Variables

- allocation is done by subtracting from esp

subroutine template

```
push ebp
mov  ebp, esp
sub  esp, N_BYTES

...

mov  esp, ebp
pop  ebp
ret
```

stack layout

var. 2	esp, ebp-8
var. 1	ebp-4
ebp	ebp
ret. addr.	ebp+4
param. 1	ebp+8
param. 2	ebp+12

Example: Factorial (C)

```
int y;

void fact(int k)
{
  register int i;

  y = 1;
  for (i = k; i > 1; i--)
      y = y * i;
}
```


Example: Factorial (C)

```
int fact(int k)
{
    int y;
    register int i;

    y = 1;
    for (i = k; i > 1; i--)
        y = y * i;
    return y;
}
```

Example: Factorial

```
segment .text
global fact

fact:
    push ebp
    mov  ebp, esp
    sub  esp, 4

    mov  dword [ebp-4], 1
    mov  ecx, [ebp+8]
```

```
back:
    mov  eax, [ebp-4]
    mul  ecx
    mov  [ebp-4], eax
    dec  ecx
    cmp  ecx, 1
    jne  back

    mov  eax, [ebp-4]
    mov  esp, ebp
    pop  ebp
    ret
```


Calling Assembly from C

- to call an Assembly function from C:
- in Assembly file: declare function as `global`
- in C file: declare the prototype

Example: Factorial

```
int fact(int k);

int main(void)
{
    int x, y;

    ...
    y = fact(x);
    ...
}
```


References

Required Reading: Carter

- Chapter 4: Subprograms