CSE320 System Fundamentals II

x86 Assembly Language

YOUNGMIN KWON / TONY MIONE

Generating an Assembly File from C

```
gcc –S –c –O0 –fverbose-asm hello.c
```

- `-S`: generate an assembly file (`hello.s`)
- `-c`: do not link
- `-O0`: no optimization [will make the generated assembler code easier to follow]
- `–fverbose-asm`: add verbose comments

---------- hello.s

```
    .file   "hello.c"
    .section    .rodata
.LC0:
    .string "hello world"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $.LC0, %edi
    call    puts
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
    .section    .note.GNU-stack,"",@progbits
```

---------- hello.c

```c
#include <stdio.h>
int main()
{
    printf("hello world\n");
    return 0;
}
```

x86 Assembly

Two different Syntaxes

- Intel Syntax: op dst, src
  - `movl eax, 1` move 1 to eax register.
  - `addl eax, ebx` eax = eax + ebx
- AT&T (GAS) Syntax: op src, dst
  - `movl 1, %eax` 1 → eax
  - `addl %eax, %ebx` ebx = eax + ebx

[Intel Architecture Reference Manuals]

Assembler Directives for Sections

`.text`
- Instructions/program code are placed here

`.data`
- Initialized read/write data are defined here

`.section .rodata`
- Initialized read only data are defined here

`.comm symbol, length, alignment`
- Uninitialized data are allocated in the bss section

`.local name`
- Makes a name a local symbol
- `.lcomm = .local + .comm`

More Assembler Directives

`.ascii "string" …`
- Define strings without the terminal zero

`.string "string" …`
- Define null-terminated strings

`.byte, .int, .long, .quad`
- Define integer numbers

`.double, .float`
- Define floating point numbers

`.align`
- Pad the location counter to a particular storage boundary.

`.size`
- Set the size associated with a symbol

AT&T Assembly Format

General format:

- operation source, destination
- e.g. `movb $0x05, %al`

Operation Suffixes

- Instructions are suffixed with
  `b`: byte, `s`: short (2 byte int or 4 byte float), `w`: word (2 byte), `l`: long(4 byte int or 8 byte float), `q`: quad (8 byte), `t`: ten byte (10 byte float)

Prefixes

- `%` for registers, `$` for constant numbers (literals/immediates)

Literals

Integers

- `24, 0b1010, 0x4a, 074`

Floating point numbers

- `0.1, 1.2e3`

Strings

- `"abc\n"`

Characters

- `'a', '\n'`

Registers

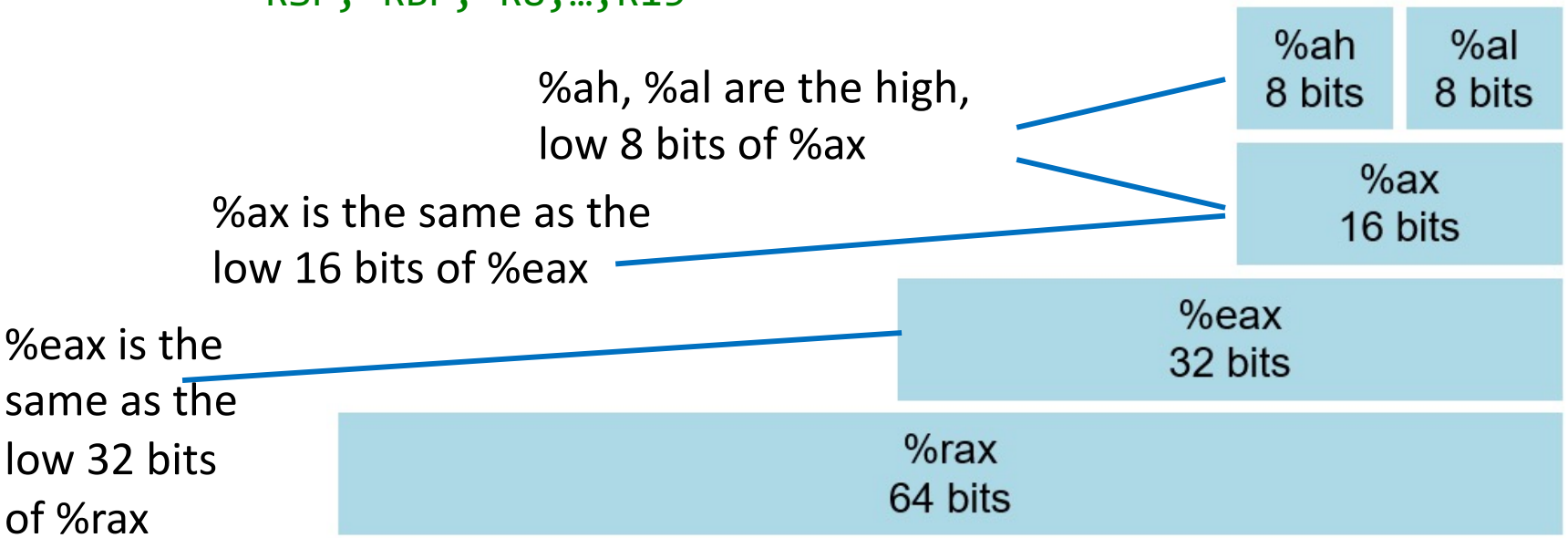
```
8 bit:  AH, AL, BH, BL, CH, CL, DH, DL, R8B,…,R15B

16 bit: AX, BX, CX, DX, SI, DI, SP, BP, R8W,…,R15W

32 bit: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP,
        R8D,…,R15D

64 bit: RAX, RBX, RCX, RDX, RSI, RDI,
        RSP, RBP, R8,…,R15
```

ESP: stack pointer



Addressing Operand

Syntax

- `segment:displacement(base register, offset register, scalar multiplier)`
- base register + offset register * scalar multiplier + displacement (ignoring segment)
- Either or both of numeric parameters can be omitted
- Either of the register parameters can be omitted

Example

- `movl -5(%rbp, %rsi, 4), %eax` `#load [rbp + rsi * 4 - 5] into eax`
- `movl -5(%rbp), %eax` `#load [rbp - 5] into eax`
- `leaq 8(%rbx, %rcx, 2), %rax` `#load rcx * 2 + rbx + 8 into rax`

load effective address

Move and Stack Manipulation Instructions

mov src, dst

- At least one of src or dst must be a register
- `e.g. movl $0, %eax`

push src — stack

- `e.g. pushl %eax`
- `eqv: subq $4, %rsp; movl %eax (%rsp)`

pop dst — stack

- `e.g. popq %rax`
- `eqv: movq (%rsp) %rax; addq $8, %rsp`

leave — after leave → return

- `movq %rbp, %rsp`
  `popq %rbp`

Arithmetic Instructions

add src, dst

- e.g. addq $2, %rax # rax = rax + 2

sub src, dst

- e.g. subq %rbx, %rax # rax = rax – rbx

mul arg

- e.g. mulw %bx # bx * ax -> dx (high 16bits), ax (lower 16bits)

div arg

- e.g. divl %ebx
  # (edx * 2^32 + eax) / ebx -> eax,
  # (edx * 2^32 + eax) mod ebx -> edx

Logical Instructions

bitwise

and src, dst

- `e.g. andl $0xf, %eax # eax &= 0xf`

10 01 ⇒ 00

or src, dst

- `e.g. orl $0xf, %eax # eax |= 0xf`

10 11 ⇒ 11

not dst

- `e.g. notq %rax # rax = ~rax`

1 ⇒ 0

xor src, dst

- `e.g. xorw %ax, %ax # ax = ax xor ax`

Flags

ZF (zero flag)

- Set if the result is 0

SF (sign flag)

- Set if the MSB of the result is 1

OF (overflow flag)

- Set when overflow occurred (8 + 8 → 16 in 4bit)
- Positive num op Positive num → Negative num
- Negative num op Negative num → Positive num

P+P→ N
N+N→ P
P-N → N
N-P → P

Compare and Branch Instructions

cmp arg1 arg2

- `cmpq $2, %rax`

```
# ZF = iff %rax - 2 == 0
# SF = iff MSB of %rax - 2 == 1
# OF = iff overflow occurs
```

test arg1 arg2

- `testq $5, %rax`

```
# ZF = iff %rax & 5 == 0
# SF = iff MSB of %rax & 5 == 1
```

Compare and Branch Instructions

JE, JZ, JNE, JNZ, JG, JGE, JL, JLE

- `jne label` Jump not equal
  `# jump if ZF == 0`
- `jg label`
  `# jump if SF == OF and ZF == 0`
  - `cmp -4, 4 => OF=1, SF=1 on 4bit machines`
  - `cmp 2, -1 => OF=1, SF=0 on 4bit machines`

Call Instructions

call label

- e.g. `call 0x1234`
- Equivalent to

```
pushq %rip
movq 0x1234, %rip
```

ret — When finish

- e.g. `ret`
- Equivalent to

```
popq %rip
```

gcd.c

```c
#include <stdio.h>
long gcd(long x, long y)
{
    while (x != y) {
        if (x > y)
            x -= y;
        else
            y -= x;
    }
    return x;
}
```

gcd.s

```
    .text
    .globl  gcd
    .type   gcd, @function
gcd:
    pushq   %rbp                #
    movq    %rsp, %rbp          #,
    movq    %rdi, -8(%rbp)      # x, x
    movq    %rsi, -16(%rbp)     # y, y
    jmp .L2
.L4:
    movq    -8(%rbp), %rax      # x, tmp61
    cmpq    -16(%rbp), %rax     # y, tmp61
    jle .L3                     #,
    movq    -16(%rbp), %rax     # y, tmp62
    subq    %rax, -8(%rbp)      # tmp62, x
    jmp .L2                     #
.L3:
    movq    -8(%rbp), %rax      # x, tmp63
    subq    %rax, -16(%rbp)     # tmp63, y
.L2:
    movq    -8(%rbp), %rax      # x, tmp64
    cmpq    -16(%rbp), %rax     # y, tmp64
    jne .L4                     #,
    movq    -8(%rbp), %rax      # x, D.2060
    popq    %rbp                #
    ret
```

----------gcd.c

```c
long print()
{
    long a = 24, b = 30;
    long c = gcd(a, b);
    printf("gcd(%ld, %ld) = %ld\n",
           a, b, c);
    return 0;
}
```

----------gcd.s

```
    .section    .rodata
.LC0:
    .string "gcd(%ld, %ld) = %ld\n"
    .text
    .globl  print
    .type   print, @function
```

```
print:
    pushq   %rbp                #
    movq    %rsp, %rbp          #,
    subq    $32, %rsp           #,
    movq    $24, -24(%rbp)      #, a
    movq    $30, -16(%rbp)      #, b
    movq    -16(%rbp), %rdx     # b, tmp62
    movq    -24(%rbp), %rax     # a, tmp63
    movq    %rdx, %rsi          # tmp62,
    movq    %rax, %rdi          # tmp63,
    call    gcd                 #
    movq    %rax, -8(%rbp)      # tmp64, c
    movl    $.LC0, %eax         #, D.2054
    movq    -8(%rbp), %rcx      # c, tmp65
    movq    -16(%rbp), %rdx     # b, tmp66
    movq    -24(%rbp), %rsi     # a, tmp67
    movq    %rax, %rdi          # D.2054,
    movl    $0, %eax            #,
    call    printf              #
    movl    $0, %eax            #, D.2055
    leave
    ret
```

Questions?