CSE320 System Fundamentals II
x86 Assembly Language

YoungMin Kwon

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
  - `-fverbose-asm`: add verbose comments

```
----------------------- hello.s
    .data
g:
    .long   1
    .section .rodata
.LC0:
    .string "hello world %d, %d\n"
    .text
    .globl  main
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
# hello.c:4:     int a = 2;
    movl    $2, -4(%rbp)
# hello.c:5:     printf("hello world %d, %d\n", g, a);
    movl    g(%rip), %eax
    movl    -4(%rbp), %edx
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
# hello.c:6:     return 0;
    movl    $0, %eax
# hello.c:7: }
    leave
    ret
```

```
----------------------- hello.c
#include <stdio.h>
static int g = 1;
int main() {
    int a = 2;
    printf("hello world %d, %d\n",
           g, a);
    return 0;
}
```

x86 Assembly

- Two different Syntaxes
  - Intel Syntax: op dst, src
    - `movl eax, 1` `# eax = 1`
    - `addl eax, ebx` `# eax = eax + ebx`
  - AT&T (GAS) Syntax: op src, dst
    - `movl $1, %eax` `# eax = 1`
    - `addl %eax, %ebx` `# ebx = ebx + eax`

Assembler Directives for Sections

- `.text`
  - Instruction codes are defined here
- `.data`
  - Initialized read/write data are defined here
- `.section .rodata`
  - Initialized read only data are defined here
- `.comm`
  - Uninitialized data are allocated in the bss section
- `.local name`
  - Makes a name a local symbol
  - `.lcomm = .local + .comm`

```
    .data
g:
    .long   1
    .section .rodata
.LC0:
    .string "hello world %d, %d\n"
    .text
    .globl  main
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $2, -4(%rbp)

    movl    g(%rip), %eax
    movl    -4(%rbp), %edx
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT

    movl    $0, %eax

    leave
    ret
```

More Assembler Directives

- `.string "string" …`
  - Defines a null-terminated string
  - `.ascii "string"`: defines a string without the null-terminator
- `.byte, .int, .long, .quad`
  - Define integer numbers
  - `.zero x`: allocate x bytes
- `.double, .float`
  - Define floating point numbers
- `.align`
  - Pad the location counter to a particular storage boundary

```
    .data
g:
    .long   1
    .section .rodata
.LC0:
    .string "hello world %d, %d\n"
    .text
    .globl  main
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $2, -4(%rbp)

    movl    g(%rip), %eax
    movl    -4(%rbp), %edx
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT

    movl    $0, %eax

    leave
    ret
```

AT&T Assembly Format

- General format:
  - Instruction 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

```
    .data
g:
    .long   1
    .section .rodata
.LC0:
    .string "hello world %d, %d\n"
    .text
    .globl  main
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $2, -4(%rbp)

    movl    g(%rip), %eax
    movl    -4(%rbp), %edx
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT

    movl    $0, %eax

    leave
    ret
```

Three Categories of Instructions

- Data transfer
  - Constants, registers, memory
- Data processing
  - Arithmetic and logical operators
- Control flow
  - Flags, jump and call operators

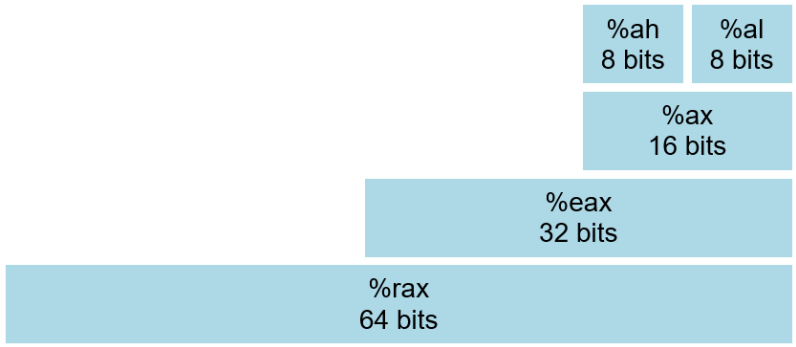
Literals

- Integers
  - decimal: 24, binary: 0b1010, hexadecimal: 0x4a, octal: 074
- Floating point numbers
  - `0.1, 1.2e3`
- Strings
  - `"abc\n"`
- Characters
  - `'a', '\n'`

```
    .data
g:
    .long   1
    .section .rodata
.LC0:
    .string "hello world %d, %d\n"
    .text
    .globl  main
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $2, -4(%rbp)

    movl    g(%rip), %eax
    movl    -4(%rbp), %edx
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT

    movl    $0, %eax

    leave
    ret
```

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




```
    .data
g:
    .long   1
    .section .rodata
.LC0:
    .string "hello world %d, %d\n"
    .text
    .globl  main
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $2, -4(%rbp)

    movl    g(%rip), %eax
    movl    -4(%rbp), %edx
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT

    movl    $0, %eax

    leave
    ret
```

Addressing Operand

- Syntax

```
segment : displacement ( base reg , offset reg , scalar multiplier )
```

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

- Example

```
# load from memory address
movl -5(%rbp, %rsi, 4), %eax  # load [rbp + rsi * 4 - 5] to eax
movl -5(%rbp), %eax           # load [rbp - 5] to eax

# lea: load effective address
leaq 8(%rbx, %rcx, 2),  %rax  # copy rbx + rcx * 2 + 8 to rax
```

Move Instructions

- mov src, dst
  - Copy from src to dst
  - Examples

```
# copy 0 to eax
movl  $0, %eax

# copy the address of .LC0 to rax
leaq  .LC0(%rip), %rax

# copy byte to long, extend zero
movzbl %al, %eax

# copy byte to long, extend the sign of al
movsbl %al, %eax
```

```
    .data
g:
    .long   1
    .section .rodata
.LC0:
    .string "hello world %d, %d\n"
    .text
    .globl  main
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $2, -4(%rbp)

    movl    g(%rip), %eax
    movl    -4(%rbp), %edx
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT

    movl    $0, %eax

    leave
    ret
```

Stack Manipulation Instructions

- push src
  - Push src to the stack

```
pushq %rax # push rax to the stack
# equivalent to
subq $8, %rsp
movq %rax (%rsp)
```

- pop dst
  - Pop from the stack and copy the result to dst

```
popq %rax # pop from the stack to rax
# equivalent to
movq (%rsp) %rax
addq $8, %rsp
```

```
    .data
g:
    .long   1
    .section .rodata
.LC0:
    .string "hello world %d, %d\n"
    .text
    .globl  main
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $2, -4(%rbp)

    movl    g(%rip), %eax
    movl    -4(%rbp), %edx
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT

    movl    $0, %eax

    leave
    ret
```

Arithmetic Instructions

- Addition and subtraction
  - add src, dst
  - sub src, dst

```
addq $2,   %rax # rax = rax + 2
subq %rbx, %rax # rax = rax - rbx
```

- Multiplication, division and modulo
  - mul arg
  - div arg

```
mulw %bx  # bx * ax -> dx:ax
          # (dx:ax = 2^16 * dx + ax,
          #  dx higher 16 bits,
          #  ax lower  16 bits)
divl %ebx # edx:eax / ebx -> eax,
          # edx:eax % ebx -> edx
```

```
    .data
g:
    .long   1
    .section .rodata
.LC0:
    .string "hello world %d, %d\n"
    .text
    .globl  main
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $2, -4(%rbp)

    movl    g(%rip), %eax
    movl    -4(%rbp), %edx
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT

    movl    $0, %eax

    leave
    ret
```

Arithmetic Instructions

```
void arith(int x, int y) {
    int a;
    a = x + y;
    a = x - y;
    a = x * y;
    a = x / y;
}

arith:
        pushq   %rbp    #
        movq    %rsp, %rbp      #,
        movl    %edi, -20(%rbp) # x, x
        movl    %esi, -24(%rbp) # y, y
# a = x + y;
        movl    -20(%rbp), %edx # x, tmp86
        movl    -24(%rbp), %eax # y, tmp87
        addl    %edx, %eax      # tmp86, tmp85
        movl    %eax, -4(%rbp)  # tmp85, a
# a = x - y;
        movl    -20(%rbp), %eax # x, tmp91
        subl    -24(%rbp), %eax # y, tmp90
        movl    %eax, -4(%rbp)  # tmp90, a
# a = x * y;
        movl    -20(%rbp), %eax # x, tmp93
        imull   -24(%rbp), %eax # y, tmp92
        movl    %eax, -4(%rbp)  # tmp92, a
# a = x / y;
        movl    -20(%rbp), %eax # x, tmp97
        cltd    # convert long to double
                # (eax -> edx:eax)
                # cqto (rax -> rdx:rax)
        idivl   -24(%rbp)       # y
        movl    %eax, -4(%rbp)  # tmp95, a
# }
        nop
        popq    %rbp    #
        ret
```

Logical Instructions

- And, Or, and Xor

```
andl $0xf, %eax # eax = eax & 0xf
orl  $0xf, %eax # eax = eax | 0xf
xorl %eax, %eax # eax = eax ^ eax
```

- 1's complement and 2's complement

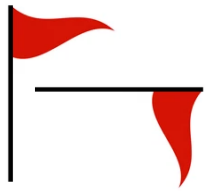
```
notq %rax # rax = ~ rax (1's complement)
negq %rax # rax = - rax (2's complement)
```

Logical Instructions

```
void logical(int x, int y) {
    int a;
    a = x & y;
    a = x | y;
    a = ~x;
}
```

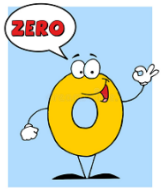
```
logical:
        pushq   %rbp    #
        movq    %rsp, %rbp      #,
        movl    %edi, -20(%rbp) # x, x
        movl    %esi, -24(%rbp) # y, y
# a = x & y;
        movl    -20(%rbp), %eax # x, tmp85
        andl    -24(%rbp), %eax # y, tmp84
        movl    %eax, -4(%rbp)  # tmp84, a
# a = x | y;
        movl    -20(%rbp), %eax # x, tmp89
        orl     -24(%rbp), %eax # y, tmp88
        movl    %eax, -4(%rbp)  # tmp88, a
# a = ~x;
        movl    -20(%rbp), %eax # x, tmp93
        notl    %eax    # tmp92
        movl    %eax, -4(%rbp)  # tmp92, a
# }
        nop
        popq    %rbp    #
        ret
```

Flags



- Flags register
  - It represents the current state of the CPU
  - Contains condition codes after arithmetic or logical operations

- 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)
  - Change sign after adding the same signed numbers or subtracting oppositely signed numbers
    - P + P → N, N + N → P, P - N → N, N - P → P

Compare Instructions

- cmp arg1 arg2
  - Compare arg2 and arg1 using subtraction without updating arg2

```
compq $2, %rax
# ZF = 1 iff %rax - 2 == 0
# SF = 1 iff MSB of %rax -2 == 1
# OF = 1 iff overflow occurred
```

- test arg1 arg2
  - Compare arg2 and arg1 using bitwise and without updating arg2

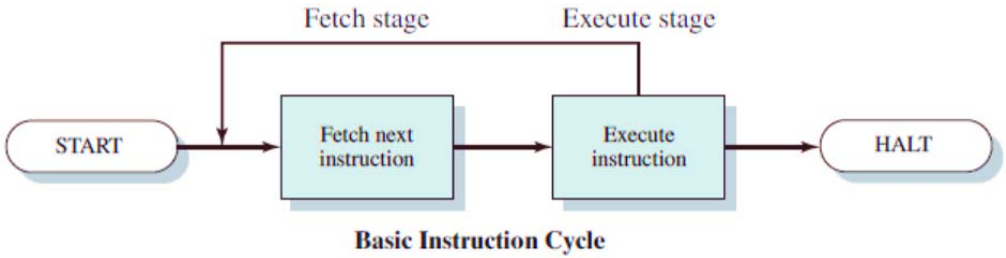
```
testq $5, %rax
# ZF = 1 iff %rax & 5 == 0
# SF = 1 iff MSB of %rax & 5 == 1
```

Compare Instructions

```
void comp(int x, int y) {
    int a;
    a = x == y;
    a = x != y;
    a = x >  y;
    a = x >= y;
    //a = x <  y;
    //a = x <= y;
}

comp:
        pushq   %rbp    #
        movq    %rsp, %rbp      #,
        movl    %edi, -20(%rbp) # x, x
        movl    %esi, -24(%rbp) # y, y
# a = x == y;
        movl    -20(%rbp), %eax # x, tmp88
        cmpl    -24(%rbp), %eax # y, tmp88
        sete    %al     #, _1
                # copy Z flag to %al
# a = x == y;
        movzbl  %al, %eax       # _1, tmp89
                # move from byte to long
        movl    %eax, -4(%rbp)  # tmp89, a
# a = x != y;
        movl    -20(%rbp), %eax # x, tmp90
        cmpl    -24(%rbp), %eax # y, tmp90
        setne   %al     #, _2
# a = x != y;
        movzbl  %al, %eax       # _2, tmp91
        movl    %eax, -4(%rbp)  # tmp91, a
# a = x >  y;
        movl    -20(%rbp), %eax # x, tmp92
        cmpl    -24(%rbp), %eax # y, tmp92
        setg    %al     #, _3
# a = x >  y;
        movzbl  %al, %eax       # _3, tmp93
        movl    %eax, -4(%rbp)  # tmp93, a
# a = x >= y;
        movl    -20(%rbp), %eax # x, tmp94
        cmpl    -24(%rbp), %eax # y, tmp94
        setge   %al     #, _4
# a = x >= y;
        movzbl  %al, %eax       # _4, tmp95
        movl    %eax, -4(%rbp)  # tmp95, a
# }
        nop
        popq    %rbp    #
        ret
```

Branch Instructions

- CPU fetches the next instruction from rip



Basic Instruction Cycle

- Unconditional jump instruction
  - JMP label: Jump to label unconditionally
  - Equivalent to copy the address of label to rip register

```
jmp     foo
# equivalent to
movq    foo, %rip
```

Branch Instructions

- Conditional jump instructions
  - JE, JZ: jump if equal
  - JNE, JNZ: jump if not equal
  - JG (JGE): jump if greater than (or equal to)
  - JL (JLE): jump if less than (or equal to)

```
jne label
# jump to label if ZF != 0

jg label
# jump to label if SF == OF and ZF == 0
#   - if overflow did not occur: SF == 0 and OF == 0
#   - if overflow did occur:     SF == 1 and OF == 1
```

Branch Instructions

```c
int max(int x, int y) {
    int a;
    if(x > y)
        a = x;
    else
        a = y;
    return a;
}
```

```
max:
        pushq   %rbp    #
        movq    %rsp, %rbp      #,
        movl    %edi, -20(%rbp) # x, x
        movl    %esi, -24(%rbp) # y, y
# if(x > y)
        movl    -20(%rbp), %eax # x, tmp84
        cmpl    -24(%rbp), %eax # y, tmp84
        jle     .L5     #,
# a = x;
        movl    -20(%rbp), %eax # x, tmp85
        movl    %eax, -4(%rbp)  # tmp85, a
        jmp     .L6     #
.L5:
# a = y;
        movl    -24(%rbp), %eax # y, tmp86
        movl    %eax, -4(%rbp)  # tmp86, a
.L6:
# return a;
        movl    -4(%rbp), %eax  # a, _6
# }
        popq    %rbp    #
        ret
```

Branch Instructions

```
int sum() {
    int i, s;
    s = 0;
    for(i = 0; i < 10; i++)
        s = s + i;
    return s;
}
```

```
sum:
        pushq   %rbp            #
        movq    %rsp, %rbp      #,
# s = 0;
        movl    $0, -8(%rbp)    #, s
# for(i = 0; i < 10; i++)
        movl    $0, -4(%rbp)    #, i
# for(i = 0; i < 10; i++)
        jmp     .L9             #
.L10:
# s = s + i;
        movl    -4(%rbp), %eax  # i, tmp84
        addl    %eax, -8(%rbp)  # tmp84, s
# for(i = 0; i < 10; i++)
        addl    $1, -4(%rbp)    #, i
.L9:
# for(i = 0; i < 10; i++)
        cmpl    $9, -4(%rbp)    #, i
        jle     .L10            #,
# return s;
        movl    -8(%rbp), %eax  # s, _5
# }
        popq    %rbp            #
        ret
```

Call Instructions

- call label
  - Call the function with the label
  - Push rip (address of the next instruction) to the stack
  - Jump to the function label

```
call    foo
# equivalent to
pushq   %rip
movq    foo, %rip
```

- ret
  - return from the function
  - Pop the next instruction address from the stack and store it at the %rip register

```
ret
# equivalent to
popq    %rip
```

```
    .data
g:
    .long   1
    .section .rodata
.LC0:
    .string "hello world %d, %d\n"
    .text
    .globl  main
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $2, -4(%rbp)

    movl    g(%rip), %eax
    movl    -4(%rbp), %edx
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT

    movl    $0, %eax

    leave
    ret
```

Call Instructions

```
        .data
g:
        .long   1
        .section .rodata
.LC0:
        .string "hello world %d, %d\n"
        .text
        .globl  main
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    $2, -4(%rbp)

        movl    g(%rip), %eax
        movl    -4(%rbp), %edx
        movl    %eax, %esi
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT

        movl    $0, %eax

        leave
        ret
```

Save the caller's stack frame

New stack frame for callee

Allocate mem for local vars …

- leave
  - Deallocate mem for the callee
  - Restore the caller's stack frame

```
leave
# equivalent to
movq %rbp, %rsp
popq %rbp
```

Deallocate mem for the caller &
Restore caller's stack frame

Call Instructions

```c
int main() {
    int a;
    a = max(10, 20);
    return 0;
}
```

```
main:
        pushq   %rbp            #
        movq    %rsp, %rbp      #,
        subq    $16, %rsp       #,
# a = max(10, 20);
        movl    $20, %esi       #,
        movl    $10, %edi       #,
        call    max             #
        movl    %eax, -4(%rbp)  # tmp84, a
# return 0;
        movl    $0, %eax        #, _4
# }
        leave
        ret
```

Compile with make

- Build with make tool
  - Write Makefile as below and run make
    - make <target> like make cmplx.cmo will build the target

Target

Dependencies

```
cmplx_app.exe: cmplx.cmi cmplx.cmo cmplx_app.cmo
    ocamlc -o cmplx_app.exe cmplx.cmo cmplx_app.cmo

cmplx.cmi: cmplx.mli
    ocamlc -c cmplx.mli

cmplx.cmo: cmplx.ml
    ocamlc -c cmplx.ml

cmplx_app.cmo: cmplx_app.ml
    ocamlc -c cmplx_app.ml
```

tab char, not spaces

Compile with make (suffix rules)

```
TGT  = cmplx_app.exe
CMIS = cmplx.cmi
CMOS = cmplx.cmo cmplx_app.cmo

RM   = del  # rm in Linux,   del  in Windows
TRUE = cd . # true in Linux, cd . in Windows

.SUFFIXES:                    # reset all suffixes
.SUFFIXES: .cmi .cmo .mli .ml # suffixes to consider

.mli.cmi:; ocamlc -c $< -o $@  # how to convert .mli to .cmi
.ml.cmo:;  ocamlc -c $< -o $@  # $< : input file name,
                               # $@ : target file name

$(TGT): $(CMIS) $(CMOS)
    ocamlc -o $@ $(CMOS)

clean:
    $(RM) *.cmi | $(TRUE)
    $(RM) *.cmo | $(TRUE)
```

macro definitions

suffix rules

using macros

true even if $(RM) failed

Assignment 3

- Download compiler_base.zip and implement all TODOs
  - To compile: make
  - It will generate spl
- Finish the implementation
  - spl should print out an assembly code for the given program
    - E.g. spl test_gcd.txt
  - Save the output as an assembly file using redirection
    - ./spl test_gcd.txt > test_gcd.s
  - Compile the assembly file using gcc and run it
    - gcc test_gcd.s
    - ./a.out
- Due date: 9/22/2022
  - Upload the changed files to blackboard in a single zip file

Assignment 3

- Makefile

```
TGT  = spl
HSRC = common.h expr.h expr_opr.h list.h parser.h refobj.h scanner.h stmt.h varstore.h
OBJS = app.o common.o list.o parser.o refobj.o scanner.o varstore.o
OBJS += expr_arith.o expr_comp.o expr_num.o expr_opr.o expr_var.o
OBJS += stmt_assignment.o stmt_compound.o stmt_if.o stmt_read.o stmt_while.o stmt_write.o
LIBS = # libregex.a #none for Linux, libregex.a for Windows

RM   = rm   # rm in Linux, del in Windows
TRUE = true # true in Linux, cd . in Windows

.SUFFIXES:          # reset all suffixes
.SUFFIXES: .c .o    # suffixes to consider

# convert .c to .o
.c.o:; gcc -c $< -o $@

$(TGT): $(HSRC) $(OBJS)
    gcc -o $@ $(OBJS) $(LIBS)
```

Assignment 3

- Makefile

```
clean:
    $(RM) *.o | $(TRUE)
    $(RM) *.s | $(TRUE)

test:
    ./$(TGT) test_arith.txt > arith.s; gcc arith.s; echo "2\n3" | ./a.out
    ./$(TGT) test_comp.txt  > comp.s;  gcc comp.s;  echo "2\n3" | ./a.out
    ./$(TGT) test_gcd.txt   > gcd.s;   gcc gcd.s;   echo "9\n6" | ./a.out
    ./$(TGT) test_sum.txt   > sum.s;   gcc sum.s;   echo "10"   | ./a.out
```

Assignment 3

- Expression
  - Evaluates to be a value
  - In SPL: the result is pushed on top of the stack
  - 0, 1, x, y, z, +, -, *, /, ==, !=, >=, >, …
- Statement
  - Actions or commands to make side effects
  - Assignment, Read, Write, While, If…

```
{
    i := 0              // orange terms are expressions
    s := 0              // blue terms are statements
    while (i <= 10) {
        s := s + i
        i := i + 1
    }
    write s
}
```

Assignment 3

- Syntax of SPL

```
program
      -> stmt EOF

stmt -> stmt_assignment
      |  stmt_read
      |  stmt_write
      |  stmt_compound
      |  stmt_if
      |  stmt_while

stmt_assignment
      -> ID := expr

stmt_read
      -> READ ID

stmt_write
      -> WRITE expr
```

```
stmt_compound
      -> { stmt_list }

stmt_list
      -> stmt
      |  stmt_list stmt

stmt_if
      -> IF ( expr ) stmt ELSE stmt

stmt_while
      -> WHILE ( expr ) stmt
```

Assignment 3

- Syntax of SPL

```
expr -> expr_comp

expr_comp
     -> expr_add
     |  expr_add EQ expr_add
     |  expr_add NE expr_add
     |  expr_add LE expr_add
     |  expr_add <  expr_add
     |  expr_add GE expr_add
     |  expr_add >  expr_add
```

```
expr_add
     -> expr_mul
     |  expr_add + expr_mul
     |  expr_add - expr_mul

expr_mul
     -> expr_factor
     |  expr_mul * expr_factor
     |  expr_mul / expr_factor

expr_factor
     -> NUM
     |  ID
     |  ( expr )
```

```
//Translating expressions…
//
static void eval_num(expr_t *self) {
…
    //push the number to the stack
    printf("    pushq   $%d\n", expr->n);
}
static void eval_var(expr_t *self) {
…
    int inx = var_store_get(expr->id); //get the index of the variable
…
    printf("    movq    (VAR + %d)(%%rip), %%rax\n", inx*8);
    printf("    pushq   %%rax\n");
}

static void oper_add() {
    printf("    popq    %%rbx\n");
    printf("    popq    %%rax\n");
    printf("    addq    %%rbx, %%rax\n");
    printf("    pushq   %%rax\n");
}
static void oper_sub() {
    //TODO: implement this function
}
static void oper_mul() {
    //TODO: implement this function
}
```

```
static void oper_div() {
    //TODO: implement this function
    //use cqto to convert rax -> rdx:rax
}

static void oper_eq() {
    //TODO: implement this function
    //hint: use cmpq, sete, ... and movzbq to get the result to rax
}
static void oper_ne() {
    //TODO: implement this function
    //hint: use cmpq, setne, ... and movzbq to get the result to rax
}
static void oper_ge() {
    //TODO: implement this function
    //hint: use cmpq, setge, ... and movzbq to get the result to rax
}
static void oper_gt() {
    //TODO: implement this function
    //hint: use cmpq, setg, ... and movzbq to get the result to rax
}
static void oper_le() {
    //TODO: implement this function
    //hint: use cmpq, setle, ... and movzbq to get the result to rax
}
static void oper_lt() {
    //TODO: implement this function
    //hint: use cmpq, setl, ... and movzbq to get the result to rax
}
```

```
//Translating statements
//

typedef struct stmt_write {
    refobj_t ref;   //ref is at the beginning of stmt
    void   ( *exec  )(struct stmt *self);
    void   ( *print )(struct stmt *self);

    expr_t *expr;
} stmt_write_t;

static void exec_write(stmt_t *self) {
…
    stmt_write_t *stmt = (stmt_write_t*) self;

    //evaluate expression
    stmt->expr->eval(stmt->expr);

    //print the result
    COMMENT("write ", stmt->expr->print(stmt->expr), "");
    printf("    popq    %%rsi\n");
    printf("    leaq    ANSW(%%rip), %%rdi\n");
    printf("    movl    $0, %%eax\n");
    printf("    call    printf\n");
}
```

```
typedef struct stmt_read {
…
    char *id;
} stmt_read_t;

static void exec_read(stmt_t *self) {
…
    stmt_read_t *stmt = (stmt_read_t*) self;
    int inx = var_store_set(stmt->id); //get/create the index of the variable

    COMMENT(strmsg("read %s", stmt->id), , "");

    //print the enter: message
    printf("    leaq    ENTR(%%rip), %%rdi\n");
    printf("    movl    $0, %%eax\n");
    printf("    call    printf\n");

    //TODO: load the ADDRESS of the variable to %rsi
    //hint: look at eval_var function of expr_var.c
    //note: we need to pass the address of the var to scanf not the value

    //read a number
    printf("    leaq    ENTR_FMT(%%rip), %%rdi\n");
    printf("    movl    $0, %%eax\n");
    printf("    call    scanf\n");
}
```

```c
typedef struct stmt_assignment {
    refobj_t ref;   //ref is at the beginning of stmt
    void   ( *exec  )(struct stmt *self);
    void   ( *print )(struct stmt *self);

    char *id;
    expr_t *rhs;
} stmt_assignment_t;

static void exec_assignment(stmt_t *self) {
    ON_FALSE_EXIT(self->ref.tag == OBJ_STMT_ASSIGN,
                  strmsg("tag (%d) is not OBJ_STMT_ASSIGN", self->ref.tag));

    stmt_assignment_t *stmt = (stmt_assignment_t*) self;
    int inx = var_store_set(stmt->id); //get/create the index of the variable

    //eval rhs
    //TODO: evaluate stmt->rhs
    //hint: use stmt->rhs->eval

    //assignment
    COMMENT(strmsg("%s := ", stmt->id), stmt->rhs->print(stmt->rhs), "");
    //TODO: copy the result of rhs to the variable
    //hint: look at eval_var function of expr_var.c

}
```

```
typedef struct stmt_if {
…   expr_t *cond;
    stmt_t *then_stmt;
    stmt_t *else_stmt;
} stmt_if_t;

static void exec_if(stmt_t *self) {
…   int label_else = label_new();
    int label_exit = label_new();

    //evaluate condition
    //TODO: evaluate stmt->cond
    //hint: use stmt->cond->eval

    //check the condition
    //TODO: jump to label_else if the evaluation result is 0
    //hint: use label_str function

    //execute then_stmt
    //TODO: execute stmt->then_stmt and jump to label_exit
    //hint: use label_str function and stmt->then_stmt->exec

    //execute else_stmt
    //TODO: print label_else and execute stmt->else_stmt
    //hint: use label_str function and stmt->else_stmt->exec

    //label_exit
    //TODO: print label_exit
    //hint: use label_str function
}
```

```
typedef struct stmt_while {
…
    expr_t *cond;
    stmt_t *loop_stmt;
} stmt_while_t;

static void exec_while(stmt_t *self) {
…
    //evaluate condition
    //TODO: print label_test and evaluate stmt->cond
    //hint: use label_str function and stmt->cond->eval

    //jump to label_exit if false
    //TODO: jump to label_exit if the evaluation result is 0
    //hint: use label_str function

    //execute the body
    //TODO: execute stmt->loop_stmt
    //hint: use stmt->loop_stmt->exec

    //end the while loop
    //TODO: jump to label_test
    //hint: use label_str function

    //TODO: print the label_exit
    //hint: use label_str function
}
```

```
$ gcc *.c -o spl

$ cat sum.txt
//sum.txt
{
    i := 0
    s := 0
    while (i <= 10) {
        s := s + i
        i := i + 1
    }
    write s
}

$ ./spl sum.txt > sum.s

$ gcc sum.s

$ ./a.out
answer: 55

$ cat sum.s
```

```
    .global main
    .text
main:
    pushq    %rbp

    # 0
    pushq   $0

    # i := 0
    popq    %rax
    movq    %rax, (VAR + 0)(%rip)

    # 0
    pushq   $0

    # s := 0
    popq    %rax
    movq    %rax, (VAR + 8)(%rip)
```

```
label_000:
    # i
    movq    (VAR + 0)(%rip), %rax
    pushq   %rax

    # 10
    pushq   $10

    # i <= 10
    popq    %rbx
    popq    %rax
    cmpq    %rbx, %rax
    setle   %al
    movzbq  %al, %rax
    pushq   %rax

    # while ( i <= 10 )
    popq    %rax
    cmpq    $0, %rax
    je      label_001

    # s
    movq    (VAR + 8)(%rip), %rax
    pushq   %rax
    # i
    movq    (VAR + 0)(%rip), %rax
    pushq   %rax

    # s + i
    popq    %rbx
    popq    %rax
    addq    %rbx, %rax
    pushq   %rax
    # s := s + i
    popq    %rax
    movq    %rax, (VAR + 8)(%rip)

    # i
    movq    (VAR + 0)(%rip), %rax
    pushq   %rax
    # 1
    pushq   $1
    # i + 1
    popq    %rbx
    popq    %rax
    addq    %rbx, %rax
    pushq   %rax
    # i := i + 1
    popq    %rax
    movq    %rax, (VAR + 0)(%rip)

    jmp     label_000

    # end while ( i <= 10 ) ...
label_001:
```

```
    # s
    movq    (VAR + 8)(%rip), %rax
    pushq   %rax
    # write s
    popq    %rsi
    leaq    ANSW(%rip), %rdi
    movl    $0, %eax
    call    printf
    movq    $0, %rax

    popq    %rbp
    ret

    .section .rodata
ANSW:       .string "answer: %ld\n"
ENTR:       .string "enter: "
ENTR_FMT:   .string "%ld"

    .data
VAR:        .zero 16
```