# AT&T x86-64 Assembly and C

DS2040, Satyajit Das

January 30, 2025

## Activities

### 1. AT&T Syntax vs. Intel Syntax

- AT&T: `movq source, destination`.
- Registers prefixed with %.

### 2. Create a Simple Assembly File `mov_demo.s`

```
touch mov_demo.s
```

### 3. Write a Data Movement Program (`mov_demo.s`):

```
        .section .data
msg:
        .asciz "Data movement demo\n"

        .section .text
        .global _start

_start:
        # Move immediate values into registers
        movq $10, %rax        # %rax = 10
        movq $20, %rbx        # %rbx = 20

        # Move between registers
        movq %rax, %rcx       # %rcx = 10

        # Move data from memory to register
        lea msg(%rip), %rdx   # %rdx = address of msg

        # Exit syscall
        movq $60, %rax        # 60 = exit syscall
        xorq %rdi, %rdi       # %rdi = 0 (exit status)
        syscall
```

### 4. Compile and Run

```
gcc -nostdlib -o mov_demo mov_demo.s
./mov_demo
```

(Program exits immediately; you can verify no errors.)

### 5. Examine with `objdump` or `gdb`

```
objdump -d mov_demo
```

### 6. Discussion

- Moving data: immediate to register, register to register, memory to register.
- Signed vs. unsigned considerations.

# 1 Part 2: Jumps and Branches (30 minutes)

### Learning Objectives

- Use conditional/unconditional jumps in assembly.
- Understand how flags and condition codes affect branching.

### Activities

#### 1. Create a Demo File

```
touch jumps_demo.s
```

#### 2. Write a Program Demonstrating Branches (jumps_demo.s):

```
        .section .text
        .global _start

_start:
        movq $0, %rax         # Clear rax
        movq $5, %rbx         # rbx = 5

        cmpq $0, %rbx         # Compare rbx with 0
        je equal_label        # Jump if rbx == 0
        jg greater_label      # Jump if rbx > 0 (signed)

less_label:
        # If rbx < 0
        movq $1, %rax
        jmp end_label

equal_label:
        # If rbx == 0
        movq $2, %rax
        jmp end_label

greater_label:
        # If rbx > 0
        movq $3, %rax

end_label:
        # Exit
        movq %rax, %rdi
        movq $60, %rax        # exit code
        syscall
```

#### 3. Compile and Run

```
gcc -nostdlib -o jumps_demo jumps_demo.s
./jumps_demo
echo $?
```

You should see an exit code of 3 for `rbx > 0`.

4. **Discussion**

- Condition codes: `je`, `jg`, `jl`, `jne`, etc.
- Unsigned vs. signed jumps: `ja`, `jb`, etc.

# 2 Part 3: Arithmetic Operations (30 minutes)

### Learning Objectives

- Practice `add`, `sub`, `imul`, `idiv` instructions.
- Understand register usage (%rax/%rdx) in multiplication/division.

### Activities

**1. Create `arith_demo.s`**

```
        .section .text
        .global _start

_start:
        movq $10, %rax       # rax = 10
        addq $5, %rax        # rax = rax + 5 => 15

        movq $4, %rbx        # rbx = 4
        imulq %rbx, %rax     # rax = rax * rbx => 60

        # Division: rax / rbx => quotient in rax, remainder in rdx
        movq $60, %rax
        xorq %rdx, %rdx      # clear rdx before signed division
        movq $4, %rbx
        idivq %rbx           # 60 / 4 => rax=15, rdx=0

        # Exit
        movq $60, %rax       # syscall number for exit
        xorq %rdi, %rdi      # status = 0
        syscall
```

2. **Compile and Run**

```
gcc -nostdlib -o arith_demo arith_demo.s
./arith_demo
```

3. **Discussion**

- `idivq` vs. `divq` for signed/unsigned division.
- Role of %rax and %rdx in multiplication and division.

# 3 Part 4: Procedure Calls (40 minutes)

### Learning Objectives

- Understand how to define and call assembly functions from C.

- Learn the System V AMD64 calling convention (arguments in %rdi, %rsi, %rdx, %rcx, %r8, %r9).
- Return values in %rax.

### Activities

#### 1. C Driver + Assembly Function

**2.** `main.c`

```
#include <stdio.h>

extern long asm_func(long a, long b);

int main() {
    long result = asm_func(5, 3);
    printf("Result from assembly function: %ld\n", result);
    return 0;
}
```

**3.** `asm_func.s`

```
        .text
        .global asm_func

asm_func:
        # Parameters:
        #   a -> %rdi
        #   b -> %rsi
        # Return value in %rax

        movq %rdi, %rax     # Put a in %rax
        addq %rsi, %rax     # rax = a + b
        ret
```

**4. Compile and Link**

```
gcc -c main.c       # Produces main.o
gcc -c asm_func.s   # Produces asm_func.o
gcc -o proc_demo main.o asm_func.o
./proc_demo
```

Expected output: `Result from assembly function: 8`

**5. Discussion**

- Registers for first 6 arguments: %rdi, %rsi, %rdx, %rcx, %r8, %r9.
- Return value in %rax.

## 4 Part 5: Integrating Assembly Inline in C (Optional, 25 minutes)

### Learning Objectives

- Learn to embed small AT&T instructions directly in C using `__asm__`.

## Activities

### 1. `inline_asm.c`

```
#include <stdio.h>

int main() {
    long a = 5, b = 3;
    long result;

    __asm__ volatile (
        "addq %%rbx, %%rax\n\t"
        : "=a" (result)       /* output operand */
        : "a" (a), "b" (b)    /* input operands */
    );

    printf("Result of inline addition: %ld\n", result);
    return 0;
}
```

### 2. Compile and Run

```
gcc -o inline_asm inline_asm.c
./inline_asm
```

Should output: `Result of inline addition: 8`

### 3. Discussion

- Register constraints (e.g., `"a"` binds to %rax).
- `volatile`, clobber lists, etc.

## Further Exploration

- Floating-point with SSE (%xmm0, %xmm1, etc.).
- Handling structures or large data on the stack.
- More complex syscalls (e.g., file I/O, printing to stdout).
- Using `gcc -S` on small C programs to examine generated AT&T assembly.