**CS 33 Study Guide**

\*\*\* This guide aims to solidify important concepts for CS 33. For practice we highly recommend reviewing the many examples covered in lectures, discussions, homeworks, labs, and LA worksheets and workshops. Consequently, this guide does not have any examples or practice problems. It is meant to be a GUIDE.

# **Bits, Bytes, and Integers**

1. **Representing Information as Bits**

* Everything is bits! They are just interpreted differently based on data type.
* Used to execute instructions or to represent/manipulate data.
* Understand binary and hex representations and conversion to/from decimal.

1. **Bit Level Manipulations**

* Fill in the truth tables for the following bitwise operations:

AND

| A | B | A & B |
| --- | --- | --- |
| 0 | 0 |  |
| 0 | 1 |  |
| 1 | 0 |  |
| 1 | 1 |  |

OR

| A | B | A | B |
| --- | --- | --- |
| 0 | 0 |  |
| 0 | 1 |  |
| 1 | 0 |  |
| 1 | 1 |  |

XOR

| A | B | A ^ B |
| --- | --- | --- |
| 0 | 0 |  |
| 0 | 1 |  |
| 1 | 0 |  |
| 1 | 1 |  |

* Understand the difference between ~ and ! for 32 bit numbers.
* Understand the difference between arithmetic and logical left and right shifts.
  + HINT: one is used for unsigned numbers and the other is used for signed numbers.
  + What mathematical operations do shifts perform? Are they always precise computations? HINT: think about rounding.

1. **Integers**

* Know the difference between signed and unsigned integers.
  + Unsigned integers are represented in \_\_\_\_\_\_ complement form.
  + Fill in the definitions below and know their binary representations:
    - Smallest unsigned number we can represent: \_\_\_\_\_\_\_\_\_
    - Largest unsigned number we can represent: \_\_\_\_\_\_\_\_\_
    - Smallest signed number we can represent: \_\_\_\_\_\_\_\_\_
    - Largest signed number we can represent: \_\_\_\_\_\_\_\_\_
  + Complete the following identities:
    - ~x + 1 == \_\_\_\_\_\_\_
    - TMAX + 1 == \_\_\_\_\_\_\_
    - TMAX + TMIN == \_\_\_\_\_\_
  + Understand how casting between signed and unsigned numbers works.
  + What happens when there is a mix of signed and unsigned integers in an arithmetic operation or comparison expression?
* Understand the conditions for overflow and underflow.

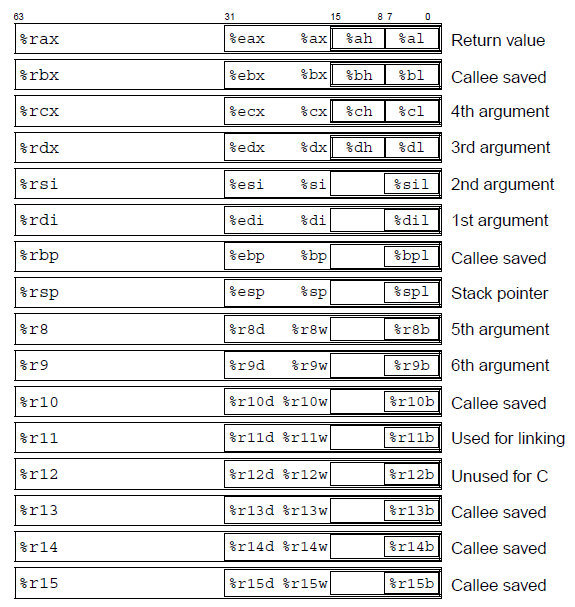
## **Studying Resources Checkpoint Checklist #1**

* Lecture 1 Notes & Examples - Bits and Bytes
* Lecture 2 Notes & Examples - Integers
* Data Lab
* Homework #1
* Week 1 LA Worksheet

# **Machine Level Programming**

1. **Assembly Basics**

* Perform arithmetic functions on either registers, memory, or literals (immediates).
* Transfer data from memory into a register and back.
* Transfer control and change instruction execution (unconditional jumps, conditional branches).
* x86 registers:

****

**\*Don’t forget %rip.**

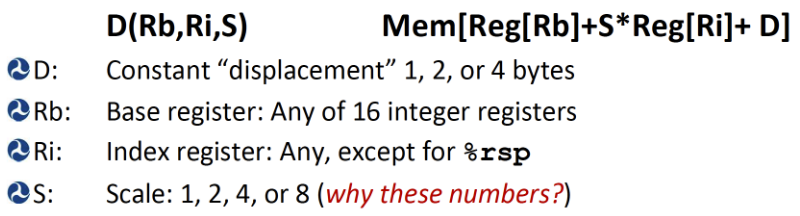
* Fill in the following table:

|  | char | short | int | long | char\* | float | double |
| --- | --- | --- | --- | --- | --- | --- | --- |
| data type |  | word |  |  |  |  |  |
| suffix | b |  |  |  |  |  |  |
| size (bytes) |  |  |  |  |  | 4 |  |

* **MOV**

| Instruction | Operand(s) | Effect | Description |
| --- | --- | --- | --- |
| mov | S, D | D ← S | Move |
| movb | S, D | D ← S | Move byte |
| movw | S, D | D ← S | Move word |
| movl | S, D | D ← S | Move double word |
| movq | S, D | D ← S | Move quad word |
| movabsq | I, R | R ← I | Move absolute quad word |

* Understand the difference between mov and lea and their addressing modes. What do the parentheses indicate in both cases?



* What are the possible source and destination operand type combinations? HINT: think registers, memory, and immediates.
* **MOVZ**

| movz | S, R | R ← zero\_extend(S) | Move w/ zero extension |
| --- | --- | --- | --- |
| movzbw | S, R | R ← zero\_extend(S) | Move w/ zero extend byte to word |
| movzbl | S, R | R ← zero\_extend(S) | Move w/ zero extend byte to double word |
| movzwl | S, R | R ← zero\_extend(S) | Move w/ zero extend word to double word |
| movzbq | S, R | R ← zero\_extend(S) | Move w/ zero extend byte to quad word |
| movzwq | S, R | R ← zero\_extend(S) | Move w/ zero extend word to double word |

* \***movzlq** doesn’t exist ⇒ happens automatically; movl having register as destination
* **MOVS**

| movs | S, R | R ← sign\_extend(S) | Move w/ sign extension |
| --- | --- | --- | --- |
| movsbw | S, R | R ← sign\_extend(S) | Move w/ sign extend byte to word |
| movsbl | S, R | R ← sign\_extend(S) | Move w/ sign extend byte to double word |
| movswl | S, R | R ← sign\_extend(S) | Move w/ sign extend word to double word |
| movsbq | S, R | R ← sign\_extend(S) | Move w/ sign extend byte to quad word |
| movswq | S, R | R ← sign\_extend(S) | Move w/ sign extend word to quad word |
| movslq | S, R | R ← sign\_extend(S) | Move w/ sign extend double word to quad word |

* \***cltq** - move w/ sign extend %eax (double word) to %rax (quad word); same as movslq %eax, %rax

1. **ARITHMETIC AND LOGICAL OPERATIONS**

* **LEAQ**

| leaq | S, D | D ← &S | Load effective address |
| --- | --- | --- | --- |

* **UNARY**

| inc | D | D ← D+1 | increment |
| --- | --- | --- | --- |
| dec | D | D ← D-1 | decrement |
| neg | D | D ← -D | negate |
| not | D | D ← -D | complement |

* **BINARY**

| add | S, D | D ← D+S | add |
| --- | --- | --- | --- |
| sub | S, D | D ← D-S | subtract |
| imul | S, D | D ← D\*S | multiply |
| xor | S, D | D ← D^S | xor |
| or | S, D | D ← D|S | or |
| and | S, D | D ← D&S | and |

* **SHIFT**

| sal | K, D | D ← D << K | left shift |
| --- | --- | --- | --- |
| shl | K, D | D ← D << K | left shift |
| sar | K, D | D ← D >>A K | arithmetic right shift |
| shr | K, D | D ← D >>L K | logical right shift |

* **SPECIAL ARITHMETIC**

| imulq | S | R[%rdx]:R[%rax] ← S\*R[%rax] | signed full multiply |
| --- | --- | --- | --- |
| mulq | S | R[%rdx]:R[%rax] ← S\*R[%rax] | unsigned full multiply |
| cqto |  | R[%rdx]:R[%rax] ← sign\_extend(R[%rax]) | convert to oct word |
| idivq | S | R[%rdx] ← R[%rdx]:R[%rax] mod S | signed divide |
| divq | S | R[%rdx] ← R[%rdx]:R[%rax]/S | unsigned divide |

1. **CONTROL**

* Examples to recall:
  + Jump tables.
  + If-else.
  + Loops (while, do-while, for).
  + Switch statements.
* **CONDITION CODES**

| Flag | Name | Description |
| --- | --- | --- |
| CF | carry flag | generated a carry out of MSB (detect overflow) |
| ZF | zero flag | yielded zero |
| SF | sign flag | yielded negative value |
| 0F | overflow flag | two’s complement overflow (positive or negative) |

* **CMP**

| cmp | S1, S2 | S2 - S1 | compare |
| --- | --- | --- | --- |
| cmpb | S1, S2 | S2 - S1 | compare byte |
| cmpw | S1, S2 | S2 - S1 | compare word |
| cmpl | S1, S2 | S2 - S1 | compare double word |
| cmpq | S1, S2 | S2 - S1 | compare quad word |

* **TEST**

| test | S1, S2 | S2 & S1 | test |
| --- | --- | --- | --- |
| testb | S1, S2 | S2 & S1 | test byte |
| testw | S1, S2 | S2 & S1 | test word |
| testl | S1, S2 | S2 & S1 | test double word |
| testq | S1, S2 | S2 & S1 | test quad word |

* **SET**

| Instruction | Operand(s) | Synonym | Effect | Condition |
| --- | --- | --- | --- | --- |
| sete | D | setz | D ← ZF | equal/zero |
| setne | D | setnz | D ← ~ZF | not equal/not zero |
| sets | D |  | D ← SF | negative |
| setns | D |  | D ← ~SF | nonnegative |
| setg | D | setnle | D ← ~(SF^OF) & ~ZF | greater (signed >) |
| setge | D | setnl | D ← ~(SF^OF) | greater or equal (signed >=) |
| setl | D | setnge | D ← SF^OF | less (signed <) |
| setle | D | setng | D ← (SF^OF) | ZF | less or equal (signed <=) |
| seta | D | setnbe | D ← ~CF & ~ZF | above (unsigned >) |
| setae | D | setnb | D ← ~CF | above or equal (unsigned >=) |
| setb | D | setnae | D ← CF | below (unsigned <) |
| setbe | D | setna | D ← CF | ZF | below or equal (unsigned <=) |

* **JUMP**

| jmp | Label |  | | | direct jump |
| --- | --- | --- | --- | --- |
| jmp | \*Operand |  | | | indirect jump |
| je | Label | jz | ZF | equal/zero |
| jne | Label | jnz | ~ZF | not equal/not zero |
| js | Label |  | SF | negative |
| jns | Label |  | ~SF | nonnegative |
| jg | Label | jnle | ~(SF^OF) & ~ZF | greater (signed >) |
| jge | Label | jnl | ~(SF^OF) | greater or equal (signed >=) |
| jl | Label | jnge | SF^OF | less (signed <) |
| jle | Label | jng | (SF^OF) | ZF | less or equal (signed <=) |
| ja | Label | jnbe | ~CF & ~ZF | above (unsigned >) |
| jae | Label | jnb | ~CF | above or equal (unsigned >=) |
| jb | Label | jnae | CF | below (unsigned <) |
| jbe | Label | jna | CF | ZF | below or equal (unsigned <=) |

* **CONDITIONAL MOVES (CMOV)**

| cmove | S, R | cmovz | ZF | equal/zero |
| --- | --- | --- | --- | --- |
| cmovne | S, R | cmovnz | ~ZF | not equal/not zero |
| cmovs | S, R |  | SF | negative |
| cmovns | S, R |  | ~SF | nonnegative |
| cmovg | S, R | cmovnle | ~(SF^OF) & ~ZF | greater (signed >) |
| cmovge | S, R | cmovnl | ~(SF^OF) | greater or equal (signed >=) |
| cmovl | S, R | cmovnge | SF^OF | less (signed <) |
| cmovle | S, R | cmovng | (SF^OF) | ZF | less or equal (signed <=) |
| cmova | S, R | cmovnbe | ~CF & ~ZF | above (unsigned >) |
| cmovae | S, R | cmovnb | ~CF | above or equal (unsigned >=) |
| cmovb | S, R | cmovnae | CF | below (unsigned <) |
| cmovbe | S, R | cmovna | CF | ZF | below or equal (unsigned <=) |

## **Studying Resources Checkpoint Checklist #2**

* Lecture 3 Notes & Examples - Machine-Level Programming I: Basics
* Lecture 4 Notes & Examples - Machine-Level Programming II: Control
* Week 2 LA Worksheet

1. **Procedures**

* **PUSH and POP**

| Instruction | Operand(s) | Effect | Description |
| --- | --- | --- | --- |
| pushq | S | R[%rsp] ← R[%rsp-8]  M[R[%rsp]] ← S | Push quad word |
| popq | D | D ← M[R[%rsp]]  R[%rsp] ← R[%rsp]+8 | Pop quad word |

* Understand the stack (which direction it grows, what affects it):
  + During a procedure call.
  + During pop, push, call, and ret instructions.
  + How and which registers interact with it.
  + In context of caller and callee saved registers.
  + Recursion.

1. **Data**

* Arrays
  + How to find elements in 1-dimensional, multi-dimensional, and multi-level arrays from a hex dump.
    - * HINT: Think Magic 8 Ball and Midterm Sample #3.
* Structs and Unions
  + How to find the size structs and unions.
    - Remember alignment rules.
  + How to find elements in structs and unions from a hex dump.

## **Studying Resources Checkpoint Checklist #3**

* Lecture 5 Notes & Examples - Machine-Level Programming III: Procedures
* Lecture 6 Notes & Examples - Machine-Level Programming IV: Data
* Bomb Lab
* Homework #2
* Homework #3
* Week 3 LA Worksheet
* Week 5 LA Worksheet
* Midterm Sample