Exam 2 Notes

Registers

Registers are small, fast storage locations directly inside the CPU that are used to hold data temporarily during the execution of programs. In the x86 architecture, which has evolved over the years from 16-bit to 32-bit (x86) and then 64-bit (x86-64 or AMD64) versions, registers have specific roles and sizes that influence their usage in various operations.

- **General-Purpose Registers (GPRs)**: Initially, x86 CPUs had 8-bit and 16-bit GPRs, but this expanded to 32-bit in the 386 and later processors, and 64-bit in x86-64 processors. These registers include:
  - `AX, BX, CX, DX` (**Accumulator, Base, Count, Data**): Used for arithmetic, data storage, loop counters, and more. In 32-bit mode, they are extended to `EAX, EBX, ECX,` and `EDX,` and further to `RAX, RBX, RCX,` and `RDX` in 64-bit mode.
  - `SI, DI, BP, SP` (**Source Index, Destination Index, Base Pointer, Stack Pointer**): Used for string operations, stack management, and memory addressing. Extended to `ESI, EDI, EBP, ESP` in 32-bit, and `RSI, RDI, RBP, RSP` in 64-bit.
  - **8 new general-purpose registers** (`R8` to `R15`) are available in x86-64, providing additional flexibility.
- **Segment Registers**: Used in real mode and protected mode for memory segmentation, helping with memory management by dividing memory into smaller segments.
  - `CS, DS, ES, FS, GS, SS` (**Code, Data, Extra, more segment registers**): They are primarily used to hold the segments' base addresses used by the CPU to access memory.
- **Instruction Pointer (IP)**: The IP (or `EIP` in 32-bit, `RIP` in 64-bit) register points to the next instruction to be executed. It's automatically updated by the CPU.
- **Flag Registers**: The `FLAGS` register (`EFLAGS` in 32-bit, `RFLAGS` in 64-bit) contains flags that indicate the status of the processor and the outcome of various operations, such as the Zero flag, Carry flag, etc.
- **Control Registers**: Used in protected mode to control operations such as memory management, task switching, and more. `CR0, CR2, CR3,` and `CR4` are examples.
- **MMX, XMM, and YMM Registers**: Used for SIMD (Single Instruction, Multiple Data) operations to perform parallel processing on multiple data points. These are beyond the basic x86 registers and are used for advanced multimedia and arithmetic operations.

Assembly Instructions

In the context of assembly language, there are common operations that can be used for registers and memory addresses. Below are some common operations found in assembly.

Instruction	Description	Syntax
ADD	Performs addition operation	ADD destination, source
CALL	Calls a procedure	CALL procedure_label
CMP	Compares two values	CMP operand1, operand2
DEC	Decrements the value of a register/memory	DEC destination
INC	Increments the value of a register/memory	INC destination
JE	Jump if equal (zero flag set)	JE label
JNE	Jump if not equal (zero flag clear)	JNE label
JMP	Unconditional jump to a label	JMP label
LEA	Computes address of memory operand and stores in register	LEA register, memory_reference
MOV	Transfers data from one location to another	MOV destination, source
POP	Pops a value from the stack	POP destination
PUSH	Pushes a value onto the stack	PUSH source
RET	Returns from a procedure	RET
SUB	Performs subtraction operation	SUB destination, source
TEST	Tests bits by performing a bitwise AND	TEST operand1, operand2

Signed Data

When dealing with signed data, the CPU uses the sign flag (SF), overflow flag (OF), and zero flag (ZF) to determine the outcome of a comparison. Here are some of the conditional jump instructions tailored for signed comparisons:

- JG (**Jump if Greater**): Jumps if the result of a subtraction is positive, and no overflow occurs (SF=OF and ZF=0).
- JL (**Jump if Less**): Jumps if the result is negative considering signed operands (SF $\neq$ OF).
- JE (**Jump if Equal**): Jumps if the operands are equal (ZF=1), applicable to both signed and unsigned comparisons.
- JGE (**Jump if Greater or Equal**): Jumps if a signed number is greater than or equal to another (SF=OF).
- JLE (**Jump if Less or Equal**): Jumps if a signed number is less than or equal or if the result is zero (ZF=1 or SF $\neq$ OF).

Unsigned Data

For unsigned data comparisons, the CPU relies on the carry flag (CF) and zero flag (ZF) to make jump decisions:

- JA (**Jump if Above**): Jumps if the first operand is greater than the second operand in an unsigned comparison (CF=0 and ZF=0).
- JB (**Jump if Below**): Jumps if the first unsigned operand is less than the second (CF=1).
- JAE (**Jump if Above or Equal**): Jumps if the first operand is greater than or equal to the second operand in an unsigned comparison (CF=0).
- JBE (**Jump if Below or Equal**): Jumps if the first operand is less than or equal to the second operand in an unsigned comparison (CF=1 or ZF=1).

Assembly Instruction Example

Below is an example of some assembly instructions and their outcomes. Assume the register %rax holds the

value 10, and `%rcx` holds the value 4.

```
    leal (%rax, %rax, 2), %rdx :  %rdx = %rax + 2(%rax) = 3(%rax) = 3(10) = 30
     leal 4 (%rcx, %rax), %rdx :  %rdx = 4 + %rcx + %rax = 4 + 4 + 10 = 18
        leal (, %rcx, 4), %rdx :  %rdx = 0 + 4(%rcx) = 4(4) = 16
  leal 4 (%rax, %rcx, 8), %rdx :  %rdx = 4 + %rax + 8(%rcx) = 4 + 10 + 8(4) = 14 + 32 = 46
```

Please note, the parenthesis in the RHS of the computations above are being used as multiplication symbols.

Memory Addressing

Memory addressing is the scheme used by a CPU to locate and access data in the computer's memory. Each byte in memory has a unique address, much like houses on a street. Instructions in assembly language use these addresses to specify where data should be read from or written to.

Operands

In the context of assembly language, an operand can be considered as an argument to an instruction that specifies what data is to be operated on. Operands can be immediate values (directly provided in the instruction), register values (which hold a small amount of data within the CPU for quick access), or memory addresses (which point to locations in RAM).

Register Values And Arithmetic Computations

Registers are used for a variety of purposes in assembly language, including but not limited to holding operands for arithmetic computations. Here's how register values are typically involved in arithmetic computations with memory addresses:

- **Register as Direct Operand**: A register can hold one of the operands for an arithmetic operation. For example, `ADD EAX, EBX` adds the contents of `EBX` to `EAX` and stores the result in `EAX`.
- **Memory Addressing Modes**: When combined with arithmetic operations, several addressing modes can be used to refer to memory addresses:
  - **Immediate Addressing**: Using a literal number. For instance, `ADD EAX, 5` adds 5 to the contents of `EAX`.
  - **Direct Addressing**: Using a direct memory address. For example, `ADD EAX, (0x0040)` adds the value at memory address `0x0040` to `EAX`.
  - **Indirect Addressing**: Using a register to hold the memory address. For example, `ADD EAX, (EBX)` adds to `EAX` the value at the memory address contained in `EBX`.
  - **Based Addressing with Displacement**: Using a register plus an offset. For example, `ADD EAX, (EBX + 8)` adds to `EAX` the value at the memory address `EBX` plus 8 bytes.
- **Computation Results**: After an arithmetic operation is performed, the result can be stored back in a register or in a memory location, depending on the instruction used.

CPUs use memory addressing to:

- Retrieve instructions to be executed.
- Access data operands for instruction execution.
- Store the results of computations.

Memory Addressing Example

Below are some simple examples of memory addressing and arithmetic operations with registers and memory addressing:

```
addl 16 (%ebp), %ecx :  Reg[ecx] = Reg[ecx] + Mem[Reg[ebp]] + 16
  addq $0x11, (%rax) :  Mem[Reg[rax]] = 17 + Mem[Reg[rax]]
  subl $0x11, (%eax) :  Mem[Reg[eax]] = Mem[Reg[eax]] - 17
```

When registers are enclosed by parenthesis, this means we are accessing that register in memory. And thus, the operations must deal with the value in memory and not just the value of the register.