Bitwise ops and float representation

IEEE 32-bit float 1 sign bit, 8 exponent bits, 23 fraction bits (mantissa). Value is $M * 2^E$. 0[1]*7 is 0 for exponents (baseline is 127). Have to add 1 for Mantissa.

Miniﬂoats Same as IEEE case, but there is only four exponent bits, and 3 manitssa bits. Baseline exponent is 0[1]*3.

Assembly

Addresing modes True to its CISC nature, x86-64 supports a variety of addressing modes. An addressing mode is an expression that calculates an address in memory to be read/written to. These expressions are used as the source or destination for a mov instruction and other instructions that access memory. The code below demonstrates how to write the immediate value 0 to various memory locations in an example of each of the available addressing modes:
movl $0, 0x604892 direct (address is constant value)
movl $0, (%rax) indirect (address is in register
movl $0, -24(%rbp) indirect with displacement (address = base
movl $0, 0xc(%rsp, %rdi, 4) indirect with displacement and scaled-index (address = base
movl $0, (%rax, %rcx, 8) (special case of scaled-index, displacement assumed 0)
movl $0, 0x8(, %rdx, 4) (special case of scaled-index, base assumed 0)
movl $0, 0x4(%rax, %rcx) (special case of scaled-index, scale assumed 1)

Mov and lea Mov copies a value from source to destination. The source can be an immediate value, a register, or a memory location (expressed using one of the addressing mode expressions from above). The destination is either a register or a memory location. At most one of source or destination can be memory. The mov suffix (b, w, l, or q) indicates how many bytes are being copied (1, 2, 4, or 8 respectively). For the lea (load effective address) instruction, the source operand is a memory location (using an addressing mode from above) and it copies the calculated source address to destination. Note that lea does not dereference the source address, it simply calculates its location. This means lea is nothing more than an arithmetic operation and commonly used to calculate the value of simple linear combinations that have nothing to do with memory location.

Memory hierarchy

Registers, L1 cache SRAM, L2 cache (SRAM), Main memory (DRAM), Local secondary storage (Disks), Remote secondary storage (DFS, Web servers)
Spatial and temporal locality
Pretty realistic scenario: › 97› Cache hit: 1 cycle to access › Cache miss: 100 cycles to access ☒ What percent of your total memory access time is spent on the 3of memory accesses that are cache misses? A. <=3B. 30C. 50D. >50› Bonus discussion: How much does this change if your cache hit rate goes up slightly to 99

Answer: D

Code samples

Replace all Complete the replace_all function. The function takes a pointer to the head of a linked list, a match-identifying callback function, and a pointer to a replacement value. It traverses the linked list looking for elements matching some criteria (as identified by the callback function match), and replaces (overwrites in place) all the matching values with the value pointed to by the replace parameter. Each node of the linked list is a void* blob that consists of a next pointer followed by a generic type value that is valuesz bytes. The next pointer and value area are laid out in contiguous memory as shown in the diagram below. In this example, the values depicted as [valueB] and [valueC] were both identified as matches by the match function, and they were replaced by a value depicted as [valueX].

```
void replace_all(void *list, size_t valuesz,
bool (*match)(const void *), const void *replace)
{
 void *curr = list;
 while (curr != NULL) {
 void *value = (char*)curr + sizeof(void*);
 if (match(value)) {
 memcpy(value, replace, valuesz); }
 curr = *(void**)curr; } }
```

Minifloat Minifloat of 40: 01100010; MF of 4.5: 01001001; Add the two, and then your balance is lower precision.

has trio Write a function has_trio that takes an unsigned int and returns true if the int's bit representation contains at least one instance of at least three consecutive 1's. Example: return true for this input: 00000000000011100000000000000000
Example: return false for this input: 00100011000110110000010101000100

```
bool has_trio(unsigned int n)
{
 return (n & (n << 1) & (n >> 1)) != 0;
}
```

ASM Ex 1. Assembly.

```
<foo>:
 0x400516 <+0>: mov (%rsi),%rax
 0x400519 <+3>: lea (%rax,%rax,2),%rax
 0x40051d <+7>: mov %rax,%r8
 0x400520 <+10>: mov 0x18(%rsi),%rcx
 0x400524 <+14>: jmp 0x40052d <foo+23>
 0x400526 <+16>: movb \$0x61,(%rdx,%rcx,1)
 0x40052a <+20>: add %rdi,%rcx
 0x40052d <+23>: cmp %r8,%rcx
 0x400530 <+26>: jle 0x400526 <foo+16>
 0x400532 <+28>: retq
```

```
long foo(long arg1, long *arg2, char *arg3) {
 long end = *arg2 * 3;
 for (long i = arg2[3]; i <= end; i += arg1) {
 arg3[i] = 'a';
 }
 return end;
}
```

```
0x4004f6 <+0>: push %rbx
 0x4004f7 <+1>: mov %rsi,%rbx
 0x4004fa <+4>: mov %edx,%eax
 0x4004fc <+6>: not %eax
 0x4004fe <+8>: cmp %edi,%eax
 0x400500 <+10>: jle 0x40050b <bar+21>
 0x400502 <+12>: callq 0x4004f0 <func> //
          see signature below
 0x400507 <+17>: sub %eax,(%rbx)
 0x400509 <+19>: jmp 0x400514 <bar+30>
 0x40050b <+21>: mov %edx,%edi
 0x40050d <+23>: callq 0x4004f0 <func> //
          see signature below
 0x400512 <+28>: or %eax,(%rbx)
 0x400514 <+30>: pop %rbx
 0x400515 <+31>: retq
```

```
void bar(int arg1, int *arg2, int arg3) {
 if (~arg3 > arg1) {
 *arg2 -= func(arg1);
 } else {
 *arg2 |= func(arg3);
 }
}
```

Function call stack

The rsp register is used as the "stack pointer"; push and pop are used to add/remove values from the stack. The push instruction takes one operand: an immediate, a register, or a memory location. Push decrements rsp and copies the operand to be tompost on the stack. The pop instruction takes one operand, the destination register. Pop copies the topmost value to destination and increments rsp. It is also valid to directly adjust rsp to add/remove an entire array or a collection of variables with a single operation. Note the stack grows downward (toward lower addresses).
push rbx push value of rbx onto stack
pushq 0x3 push immediate value 3 onto stack
sub 0x10, rsp adjust stack pointer to set aside 16 more bytes
pop rax pop topmost value from stack into register rax
add 0x10, rsp adjust stack point to remove topmost 16 bytes
Call/return are using to transfer control between functions. The callq instruction takes one operand, the address of the function being called. It pushes the return address (current value of rip) onto the stack and then jumps to that address. The retq instruction pops the return address from the stack into the destination rip, thus resuming at the saved return address.
To set up for a call, the caller puts the first six arguments into registers rdi, rsi, rdx, rcx, r8, andr9 (any additional arguments are pushed onto the stack) and then executes the call instruction.
mov 0x3, rdi first arg is passed in rdi
mov 0x7, rsi second arg is passed in rsi
callq Binky transfers control to function Binky
When callee finishes, it writes the return value (if any) to rax, cleans up the stack, and use retq instruction to return control to the caller.
mov 0x0, eax write return value to rax
add 0x10, rsp deallocate stack frame
retq return from currently executing function, resume caller
The target for a branch or call instruction is most typically an absolute address that was determined at compile-time. However there are cases where the target is not known until runtime, such as a switch statement compiled into a jump table or when invoking a function pointer. For these, the target address is computed and stored in a register and the branch/call variant is used je *rax or callq *rax to read the target address from the specified register.

Code Samples 2

Make list Complete the makelist function started below that takes a generic array of elements and creates a linked list of those same elements. The function's three arguments are the base address of the array, the number of elements in the array, and the size of each element in bytes. The function returns a pointer to a linked list of cells, one cell for each element in the array. The order of the elements in the list is the same as in the array. Each list cell consists of an element of elemsz bytes and a pointer to the next cell (null if this is the last cell). The element and the pointer are laid out in contiguous memory as shown below:

```
void *make_list(void *arr, int nelems, int elemsz)
{
 void *list = NULL;
 for (int i = nelems-1; i >= 0; i--) {
 void *cell = malloc(elemsz + sizeof(void *));
 memcpy(cell, (char *)arr + i*elemsz, elemsz);
 *(void **)((char *)cell + elemsz) = list;
 list = cell;
 }
 return list;
}
```

Multi add You are to write a function that adds four 1-bit integers at the same time.

```
/* Concise version */
unsigned char multi_add(unsigned char c1,
          unsigned char c2) {
 return ((c1 &c2) << 1) | (c1 ^ c2);
 }
/* This is the same but broken down to each step
          for clarity */
unsigned char multi_add(unsigned char c1,
          unsigned char c2) {
 unsigned char twos_place_sum = c1 & c2; // i.e.,
          the carry
 twos_place_sum = twos_place_sum << 1; //
          carry needs to move over
 unsigned char ones_place_sum = c1 ^ c2;
 return twos_place_sum | ones_place_sum; //
          combine sum and carry
 }
```

Float epsilon Find the smallest x greater than the y given above (2.25) such that x + y == x. We need to add something to 1.001 * 21 that is too far to the left for the 1.001* 21 to matter. So that's 10000.000 * 21 , because the sum 10001.001* 21 will be truncated to 10000.0 * 21 , or 1.000 * 25 . Note that you don't need to know anything about rounding to solve this problem.

```
0000000000040055d <ham>:
 40055d: push %rbp
 40055e: push %rbx
 40055f: sub $0x8,%rsp
 400563: mov %rdi,%rbp
 400566: mov %rsi,%rbx
 400569: mov %rdi,%rsi
 40056c: mov $0x400674,%edi
 400571: mov $0x0,%eax
 400576: callq <scanf>
 40057b: movslq %eax,%rdx
 40057e: cmp %eax,0x0(%rbp,%rdx,4)
 400582: jg 400590
 400584: jmp 400595
 400586: movslq %eax,%rdx
 400589: movb $0x43,(%rbx,%rdx,1)
 40058d: add $0x1,%eax
 400590: cmp $0x18,%eax
 400593: jle 400586
 400595: movslq 0x10(%rbp),%rax
 400599: add %rbx,%rax
 40059c: add $0x8,%rsp
 4005a0: pop %rbx
 4005a1: pop %rbp
 4005a2: retq
```

```
int ham(char *burr, char *aaron)
{
 int george = strcmp(burr, aaron);
 if (george == -1) {
 for (unsigned int i = 0; i < 28; i++) {
 burr[i] = '\0';
 }
 } else {
 return burr[2];
 }
 return aaron[2];
}
```

Code Samples 3

Make list example Complete the make_list function. The function takes a pointer to beginning of an array of keyvalue pairs, the number of elements (number of pairs) in the array, and the size of the value field in bytes. It returns a linked list of the key-value pairs. The keys are strings whose individual chars (including a null terminating character) are stored within the array itself (not char* pointing to memory outside the array), and the values are a generic type of size valuesz bytes.
void *make_list(void *arr, size_t nelems, size_t valuesz); The key string lengths may differ, so the null terminating character marks the end of the string, and (in the following byte) the beginning of the value. Following the value's valuesz bytes is the next key string (again, all contiguous in the input array). The below diagrams show an example input and return value for that input.
The output linked list should consist of newly allocated void* "blob" nodes that consist of (in this order): a void* next pointer, the key string (stored inside the blob, not as a char* pointing to memory outside the blob), and the value. The nodes should be in the same order as they were in the array, with the end of the list marked by a NULL value in the next pointer of the last node. After you copy the data from the array, it is no longer needed and should be freed.

```
void *make_list(void *arr, size_t nelems, size_t
          valuesz)
{
 void *list_front = NULL;
 void *list_curr = NULL;
 int blobsize;
 char *arr_curr = arr;
 // loop over array
 for (size_t i = 0; i < nelems; i++) {
 // set up blob
 blobsize = sizeof(void*) + strlen(arr_curr) + 1 +
          valuesz;
 void *node = malloc(blobsize);
 memcpy((char*)node + sizeof(void*), arr_curr,
 blobsize –sizeof(void*));
 if (list_front == NULL) {
 // set up first node of list
 list_front = node;
 list_curr = node;
 *(void**)list_curr = NULL;
 } else {
 // connect list nodes (you add line(s) of code here)
 *(void**)node = NULL;
 *(void**)list_curr = node;
 list_curr = node;
 }
 arr_curr = arr_curr + blobsize –sizeof(void*);
 }
 free(arr);
 r
```

odd cols Write a function odd_cols that takes an unsigned int and returns true if every column of the input number is odd, otherwise false (i.e., returns false if any column with zero, two, or four 1's in it).

```
bool odd_cols(unsigned int n)
{
 unsigned char *ptr = (unsigned char*)&n;
 unsigned char row0, row1, row2, row3;
 row0 = ptr[0];
 row1 = ptr[1];
 row2 = ptr[2];
row3 = ptr[3];
return (unsigned char) ~(row0 ^ row1 ^ row2
          ^ row3) == 0;
}
```

```
000000000040055d <ham>:
 40055d: push %rbp
 40055e: push %rbx
 40055f: sub $0x8,%rsp
 400563: mov %rdi,%rbp
 400566: mov %rsi,%rbx
 400569: mov %rdi,%rsi
 40056c: mov $0x400674,%edi
 400571: mov $0x0,%eax
 400576: callq <scanf>
 40057b: movslq %eax,%rdx
 40057e: cmp %eax,0x0(%rbp,%rdx,4)
 400582: jg 400590
 400584: jmp 400595
 400586: movslq %eax,%rdx
 400589: movb $0x43,(%rbx,%rdx,1)
 40058d: add $0x1,%eax
 400590: cmp $0x18,%eax
 400593: jle 400586
 400595: movslq 0x10(%rbp),%rax
 400599: add %rbx,%rax
 40059c: add $0x8,%rsp
 4005a0: pop %rbx
 4005a1: pop %rbp
 4005a2: retq
```

```
char *ham(int peggy[ ], char *aaron)
{
 int george = scanf("%d", peggy);
 if (george < peggy[george]) {
 for (int i = george; i < 25; i++) {
 aaron[i] = 'C';
 }
 }
 return aaron + peggy[4];
}
```

x86 Notes

Figure 1. x86 Registers
Memory and Addressing Modes

Modern x86—compatible processors are capable of addressing up to 232 bytes of memory: memory addresses are 32—bits wide. In the examples above, where we used labels to refer to memory regions, these labels are actually replaced by the assembler with 32—bit quantities that specify addresses in memory. In addition to supporting referring to memory regions by labels (i.e. constant values), the x86 provides a flexible scheme for computing and referring to memory addresses: up to two of the 32—bit registers and a 32—bit signed constant can be added together to compute a memory address. One of the registers can be optionally pre—multiplied by 2, 4, or 8.

The addressing modes can be used with many x86 instructions (we'll describe them in the next section). Here we illustrate some examples using the mov instruction that moves data between registers and memory. This instruction has two operands: the first is the source and the second specifies the destination.

Some examples of mov instructions using address computations are:

```
mov (%ebx), %eax /* Load 4 bytes from the
          memory address in EBX into EAX. */
mov %ebx, var(,1) /* Move the contents of EBX into
          the 4 bytes at memory address var.
(Note, var is a 32—bit constant). */
mov —4(%esi), %eax /* Move 4 bytes at memory
          address ESI + (—4) into EAX. */
mov %cl, (%esi,%eax,1) /* Move the contents of CL
          into the byte at address ESI+EAX. */
mov (%esi,%ebx,4), %edx /* Move the 4 bytes of
          data at address ESI+4*EBX into EDX. */
```

Some examples of invalid address calculations include:

```
mov (%ebx,%ecx,—1), %eax /* Can only add
          register values. */
mov %ebx, (%eax,%esi,%edi,1) /* At most 2
          registers in address computation. */
```

Instructions

Machine instructions generally fall into three categories: data movement, arithmetic/logic, and control—flow. In this section, we will look at important examples of x86 instructions from each category. This section should not be considered an exhaustive list of x86 instructions, but rather a useful subset. For a complete list, see Intel's instruction set reference.

We use the following notation:

<reg32> Any 32—bit register (%eax, %ebx, %ecx, %edx, %esi, %edi, %esp, or %ebp)
<reg16> Any 16—bit register (%ax, %bx, %cx, or %dx)
<reg8> Any 8—bit register (%ah, %bh, %ch, %dh, %al, %bl, %cl, or %dl)
<reg> Any register
<mem> A memory address (e.g., (%eax), 4+var(,1), or (%eax,%ebx,1))
<con32> Any 32—bit immediate
<con16> Any 16—bit immediate
<con8> Any 8—bit immediate
<con> Any 8—, 16—, or 32—bit immediate
In assembly language, all the labels and numeric constants used as immediate operands (i.e. not in an address calculation like 3(%eax,%ebx,8)) are always prefixed by a dollar sign. When needed, hexadecimal notation can be used with the 0x prefix (e.g. $0xABC). Without the prefix, numbers are interpreted in the decimal basis.

Data Movement Instructions

mov —Move

The mov instruction copies the data item referred to by its first operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its second operand (i.e. a register or memory). While register—to—register moves are possible, direct memory—to—memory moves are not. In cases where memory transfers are desired, the source memory contents must first be loaded into a register, then can be stored to the destination memory address.
Syntax
mov <reg>, <reg>
mov <reg>, <mem>
mov <mem>, <reg>
mov <con>, <reg>
mov <con>, <mem>
Examples
mov %ebx, %eax —copy the value in EBX into EAX
movb $5, var(,1) —store the value 5 into the byte at location var
push —Push on stack

The push instruction places its operand onto the top of the hardware supported stack in memory. Specifically, push first decrements ESP by 4, then places its operand into the contents of the 32—bit location at address (%esp). ESP (the stack pointer) is decremented by push since the x86 stack grows down —i.e. the stack grows from high addresses to lower addresses.
Syntax
push <reg32>
push <mem>
push <con32>

Examples
push %eax —push eax on the stack
push var(,1) —push the 4 bytes at address var onto the stack

pop —Pop from stack

The pop instruction removes the 4—byte data element from the top of the hardware—supported stack into the specified operand (i.e. register or memory location). It first moves the 4 bytes located at memory location (%esp) into the specified register or memory location, and then increments ESP by 4.
Syntax
pop <reg32>
pop <mem>

Examples
pop %edi —pop the top element of the stack into EDI.
pop (%ebx) —pop the top element of the stack into memory at the four bytes starting at location EBX.
lea —Load effective address

The lea instruction places the address specified by its first operand into the register specified by its second operand. Note, the contents of the memory location are not loaded, only the effective address is computed and placed into the register. This is useful for obtaining a pointer into a memory region or to perform simple arithmetic operations.
Syntax
lea <mem>, <reg32>
Examples
lea (%ebx,%esi,8), %edi —the quantity EBX+8*ESI is placed in EDI.
lea val(,1), %eax —the value val is placed in EAX.

Arithmetic and Logic Instructions

add —Integer addition
The add instruction adds together its two operands, storing the result in its second operand. Note, whereas both operands may be registers, at most one operand may be a memory location.
Syntax
add <reg>, <reg>
add <mem>, <reg>
add <reg>, <mem>
add <con>, <reg>
add <con>, <mem>
Examples
add $10, %eax —EAX is set to EAX + 10
addb $10, (%eax) —add 10 to the single byte stored at memory address stored in EAX

sub —Integer subtraction

The sub instruction stores in the value of its second operand the result of subtracting the value of its first operand from the value of its second operand. As with add, whereas both operands may be registers, at most one operand may be a memory location.
Syntax
sub <reg>, <reg>
sub <mem>, <reg>
sub <reg>, <mem>
sub <con>, <reg>
sub <con>, <mem>
Examples
sub %ah, %al —AL is set to AL — AH
sub $216, %eax —subtract 216 from the value stored in EAX

inc, dec —Increment, Decrement

The inc instruction increments the contents of its operand by one. The dec instruction decrements the contents of its operand by one.
Syntax
inc <reg>
inc <mem>
dec <reg>
dec <mem>

Examples
dec %eax —subtract one from the contents of EAX
incl var(,1) —add one to the 32—bit integer stored at location var

imul —Integer multiplication

The imul instruction has two basic formats: two—operand (first two syntax listings above) and three—operand (last two syntax listings above).
The two—operand form multiplies its two operands together and stores the result in the second operand. The result (i.e. second) operand must be a register.

The three operand form multiplies its second and third operands together and stores the result in its last operand. Again, the result operand must be a register. Furthermore, the first operand is restricted to being a constant value.

Syntax
imul <reg32>, <reg32>
imul <mem>, <reg32>
imul <con>, <reg32>, <reg32>
imul <con>, <mem>, <reg32>

Examples
imul (%ebx), %eax —multiply the contents of EAX by the 32—bit contents of the memory at location EBX. Store the result in EAX.
imul $25, %edi, %esi —ESI is set to EDI * 25
idiv —Integer division

The idiv instruction divides the contents of the 64 bit integer EDX:EAX (constructed by viewing EDX as the most significant four bytes and EAX as the least significant four bytes) by the specified operand value. The quotient result of the division is stored into EAX, while the remainder is placed in EDX.
Syntax
idiv <reg32>
idiv <mem>

Examples

idiv %ebx —divide the contents of EDX:EAX by the contents of EBX. Place the quotient in EAX and the remainder in EDX.
idivw (%ebx) —divide the contents of EDX:EAS by the 32—bit value stored at the memory location in EBX. Place the quotient in EAX and the remainder in EDX.
and, or, xor —Bitwise logical and, or, and exclusive or

These instructions perform the specified logical operation (logical bitwise and, or, and exclusive or, respectively) on their operands, placing the result in the first operand location.
Syntax
and <reg>, <reg>
and <mem>, <reg>
and <reg>, <mem>
and <con>, <reg>
and <con>, <mem>
or <reg>, <reg>
or <mem>, <reg>
or <reg>, <mem>
or <con>, <reg>
or <con>, <mem>
xor <reg>, <reg>
xor <mem>, <reg>
xor <reg>, <mem>
xor <con>, <reg>
xor <con>, <mem>
Examples
and $0x0f, %eax —clear all but the last 4 bits of EAX.
xor %edx, %edx —set the contents of EDX to zero.

not —Bitwise logical not

Logically negates the operand contents (that is, flips all bit values in the operand).
Syntax
not <reg>
not <mem>

Example
not %eax —flip all the bits of EAX

neg —Negate

Performs the two's complement negation of the operand contents.
Syntax
neg <reg>
neg <mem>

Example
neg %eax —EAX is set to (— EAX)

shl, shr —Shift left and right

These instructions shift the bits in their first operand's contents left and right, padding the resulting empty bit positions with zeros. The shifted operand can be shifted up to 31 places. The number of bits to shift is specified by the second operand, which can be either an 8—bit constant or the register CL. In either case, shifts counts of greater then 31 are performed modulo 32.

Syntax
shl <con8>, <reg>
shl <con8>, <mem>
shl %cl, <reg>
shl %cl, <mem>

shr <con8>, <reg>
shr <con8>, <mem>
shr %cl, <reg>
shr %cl, <mem>

Examples
shl $1, eax —Multiply the value of EAX by 2 (if the most significant bit is 0)
shr %cl, %ebx —Store in EBX the floor of result of dividing the value of EBX by 2n where n is the value in CL. Caution: for negative integers, it is different from the C semantics of division!
Control Flow Instructions

The x86 processor maintains an instruction pointer (EIP) register that is a 32—bit value indicating the location in memory where the current instruction starts. Normally, it increments to point to the next instruction in memory begins after execution an instruction. The EIP register cannot be manipulated directly, but is updated implicitly by provided control flow instructions.

We use the notation <label> to refer to labeled locations in the program text. Labels can be inserted anywhere in x86 assembly code text by entering a label name followed by a colon. For example,

```
        mov 8(%ebp), %esi
begin:
        xor %ecx, %ecx
        mov (%esi), %eax
```

The second instruction in this code fragment is labeled begin. Elsewhere in the code, we can refer to the memory location that this instruction is located at in memory using the more convenient symbolic name begin. This label is just a convenient way of expressing the location instead of its 32—bit value.
jmp —Jump

Transfers program control flow to the instruction at the memory location indicated by the operand.
Syntax
jmp <label>

Example
jmp begin —Jump to the instruction labeled begin.

jcondition —Conditional jump

These instructions are conditional jumps that are based on the status of a set of condition codes that are stored in a special register called the machine status word. The contents of the machine status word include information about the last arithmetic operation performed. For example, one bit of this word indicates if the last result was zero. Another indicates if the last result was negative. Based on these condition codes, a number of conditional jumps can be performed. For example, the jz instruction performs a jump to the specified operand label if the result of the last arithmetic operation was zero. Otherwise, control proceeds to the next instruction in sequence.
A number of the conditional branches are given names that are intuitively based on the last operation performed being a special compare instruction, cmp (see below). For example, conditional branches such as jle and jne are based on first performing a cmp operation on the desired operands.

Syntax
je <label> (jump when equal)
jne <label> (jump when not equal)
jz <label> (jump when last result was zero)
jg <label> (jump when greater than)
jge <label> (jump when greater than or equal to)
jl <label> (jump when less than)
jle <label> (jump when less than or equal to)

Example
cmp %ebx, %eax
jle done
If the contents of EAX are less than or equal to the contents of EBX, jump to the label done. Otherwise, continue to the next instruction .
cmp —Compare

Compare the values of the two specified operands, setting the condition codes in the machine status word appropriately. This instruction is equivalent to the sub instruction, except the result of the subtraction is discarded instead of replacing the first operand.
Syntax
cmp <reg>, <reg>
cmp <mem>, <reg>
cmp <reg>, <mem>
cmp <con>, <reg>

Example
cmpb $10, (%ebx)
jeq loop

If the byte stored at the memory location in EBX is equal to the integer constant 10, jump to the location labeled loop.
call, ret —Subroutine call and return

These instructions implement a subroutine call and return. The call instruction first pushes the current code location onto the hardware supported stack in memory (see the push instruction for details), and then performs an unconditional jump to the code location indicated by the label operand. Unlike the simple jump instructions, the call instruction saves the location to return to when the subroutine completes.
The ret instruction implements a subroutine return mechanism. This instruction first pops a code location off the hardware supported in—memory stack (see the pop instruction for details). It then performs an unconditional jump to the retrieved code location.

Syntax
call <label>
ret

Calling Convention

To allow separate programmers to share code and develop libraries for use by many programs, and to simplify the use of subroutines in general, programmers typically adopt a common calling convention. The calling convention is a protocol about how to call and return from routines. For example, given a set of calling convention rules, a programmer need not examine the definition of a subroutine to determine how parameters should be passed to that subroutine. Furthermore, given a set of calling convention rules, high—level language compilers can be made to follow the rules, thus allowing hand—coded assembly language routines and high—level language routines to call one another.

In practice, many calling conventions are possible. We will describe the widely used C language calling convention. Following this convention will allow you to write assembly language subroutines that are safely callable from C (and C++) code, and will also enable you to call C library functions from your assembly language code.

The C calling convention is based heavily on the use of the hardware—supported stack. It is based on the push, pop, call, and ret instructions. Subroutine parameters are passed on the stack. Registers are saved on the stack, and local variables used by subroutines are placed in memory on the stack. The vast majority of high—level procedural languages implemented on most processors have used similar calling conventions.

The calling convention is broken into two sets of rules. The first set of rules is employed by the caller of the subroutine, and the second set of rules is observed by the writer of the subroutine (the callee). It should be emphasized that mistakes in the observance of these rules quickly result in fatal program errors since the stack will be left in an inconsistent state; thus meticulous care should be used when implementing the call convention in your own subroutines.

Stack during Subroutine Call
[Thanks to James Peterson for finding and fixing the bug in the original version of this figure!]
A good way to visualize the operation of the calling convention is to draw the contents of the nearby region of the stack during subroutine execution. The image above depicts the contents of the stack during the execution of a subroutine with three parameters and three local variables. The cells depicted in the stack are 32—bit wide memory locations, thus the memory addresses of the cells are 4 bytes apart. The first parameter resides at an offset of 8 bytes from the base pointer. Above the parameters on the stack (and below the base pointer), the call instruction placed the return address, thus leading to an extra 4 bytes of offset from the base pointer to the first parameter. When the ret instruction is used to return from the subroutine, it will jump to the return address stored on the stack.

Caller Rules

To make a subrouting call, the caller should:

Before calling a subroutine, the caller should save the contents of certain registers that are designated caller—saved. The caller—saved registers are EAX, ECX, EDX. Since the called subroutine is allowed to modify these registers, if the caller relies on their values after the subroutine returns, the caller must push the values in these registers onto the stack (so they can be restore after the subroutine returns.
To pass parameters to the subroutine, push them onto the stack before the call. The parameters should be pushed in inverted order (i.e. last parameter first). Since the stack grows down, the first parameter will be stored at the lowest address (this inversion of parameters was historically used to allow functions to be passed a variable number of parameters).
To call the subroutine, use the call instruction. This instruction places the return address on top of the parameters on the stack, and branches to the subroutine code. This invokes the subroutine, which should follow the callee rules below.
After the subroutine returns (immediately following the call instruction), the caller can expect to find the return value of the subroutine in the register EAX. To restore the machine state, the caller should:

Remove the parameters from stack. This restores the stack to its state before the call was performed.
Restore the contents of caller—saved registers (EAX, ECX, EDX) by popping them off of the stack. The caller can assume that no other registers were modified by the subroutine .
Example

The code below shows a function call that follows the caller rules. The caller is calling a function myFunc that takes three integer parameters. First parameter is in EAX, the second parameter is the constant 216; the third parameter is in the memory location stored in EBX.

```
push (%ebx) /* Push last parameter first */
push $216 /* Push the second parameter */
push %eax /* Push first parameter last */

call myFunc /* Call the function (assume C naming) */

add $12, %esp
```

Note that after the call returns, the caller cleans up the stack using the add instruction. We have 12 bytes (3 parameters * 4 bytes each) on the stack, and the stack grows down. Thus, to get rid of the parameters, we can simply add 12 to the stack pointer.
The result produced by myFunc is now available for use in the register EAX. The values of the caller—saved registers (ECX and EDX), may have been changed. If the caller uses them after the call, it would have needed to save them on the stack before the call and restore them after it.

Callee Rules

The definition of the subroutine should adhere to the following rules at the beginning of the subroutine:

Push the value of EBP onto the stack, and then copy the value of ESP into EBP using the following instructions:

```
push %ebp
mov %esp, %ebp
```

This initial action maintains the base pointer, EBP. The base pointer is used by convention as a point of reference for finding parameters and local variables on the stack. When a subroutine is executing, the base pointer holds a copy of the stack pointer value from when the subroutine started executing. Parameters and local variables will always be located at known, constant offsets away from the base pointer value. We push the old base pointer value at the beginning of the subroutine so that we can later restore the appropriate base pointer value for the caller when the subroutine returns. Remember, the caller is not expecting the subroutine to change the value of the base pointer. We then move the stack pointer into EBP to obtain our point of reference for accessing parameters and local variables.
Next, allocate local variables by making space on the stack. Recall, the stack grows down, so to make space on the top of the stack, the stack pointer should be decremented. The amount by which the stack pointer is decremented depends on the number and size of local variables needed. For example, if 3 local integers (4 bytes each) were required, the stack pointer would need to be decremented by 12 to make space for these local variables (i.e., sub $12, %esp). As with parameters, local variables will be located at known offsets from the base pointer.
Next, save the values of the callee—saved registers that will be used by the function. To save registers, push them onto the stack. The callee—saved registers are EBX, EDI, and ESI (ESP and EBP will also be preserved by the calling convention, but need not be pushed on the stack during this step).
After these three actions are performed, the body of the subroutine may proceed. When the subroutine is returns, it must follow these steps:

Leave the return value in EAX.
Restore the old values of any callee—saved registers (EDI and ESI) that were modified . The register contents are restored by popping them from the stack. The registers should be popped in the inverse order that they were pushed.
Deallocate local variables. The obvious way to do this might be to add the appropriate value to the stack pointer (since the space was allocated by subtracting the needed amount from the stack pointer). In practice, a less error—prone way to deallocate the variables is to move the value in the base pointer into the stack pointer: mov %ebp, %esp. This works because the base pointer always contains the value that the stack pointer contained immediately prior to the allocation of the local variables.
Immediately before returning, restore the caller's base pointer value by popping EBP off the stack. Recall that the first thing we did on entry to the subroutine was to push the base pointer to save its old value.
Finally, return to the caller by executing a ret instruction. This instruction will find and remove the appropriate return address from the stack.
Note that the callee's rules fall cleanly into two halves that are basically mirror images of one another. The first half of the rules apply to the beginning of the function, and are commonly said to define the prologue to the function. The latter half of the rules apply to the end of the function, and are thus commonly said to define the epilogue of the function.