Intel x86 GNU Assembler AT&T Syntax

maksimdan

Published with GitBook

Table of Contents

My Awesome Book

This file file serves as your book's preface, a great place to describe your book's content and ideas.

C to ASM Logic

Boiler

Global

```
# comments
# comments


.text

.globl _start

_start:



done:
    movl %eax, %eax     # placeholder
```

Stack

```
#assuming helper_function is not recursive
.global main_function
.equ wordsize, 4
.text

helper_function:
    #prologue
    push %ebp
    movl %esp, %ebp
    subl $3*wordsize, %esp #make space for locals
    push %ebx
    push %edi

    # arguments
    .equ arg1,         (2*wordsize) #(%ebp)
    .equ arg2,         (3*wordsize) #(%ebp)
    .equ arg3,         (4*wordsize) #(%ebp)


    # locals
    .equ loc1,      (-1*wordsize) #(%ebp)
    .equ loc2,      (-2*wordsize) #(%ebp)
```

```
    # code


    #epilogue
    pop %esi
    pop %ebx

    movl %ebp, %esp
    pop %ebp
    ret

main_function:
    #prologue
    push %ebp
    movl %esp, %ebp
    subl $3*wordsize, %esp #make space for locals
    push %ebx
    push %edi

    # arguments
    .equ arg1,         (2*wordsize) #(%ebp)
    .equ arg2,         (3*wordsize) #(%ebp)
    .equ arg3,         (4*wordsize) #(%ebp)


    # locals
    .equ loc1,      (-1*wordsize) #(%ebp)
    .equ loc2,      (-2*wordsize) #(%ebp)


    # code
    # push arguments

    call helper_function
    movl result(%ebp), %eax
    addl $num_args*wordsize, %esp  #clear arguments


    #epilogue
    movl result(%ebp), %eax     # return saved value
    pop %edi
    pop %ebx

    movl %ebp, %esp
    pop %ebp
    ret
```

Global Variable Declaration

```
# global declaration

str:
    .string "I love CS, sometimes."

x:
    .long 5
```

Defines, or Readability Substitutions

```
.equ wordsize, 4
```

Global Array and Space Allocation

```
int x[4] = {1,5,2,18};

x:
    .long 1
    .long 5
    .long 2
    .long 18


# alternatively

x:
    .rept 1000
    .long 8
    .endr


# allocating empty space

c:
   .space 3*4*wordsize
                        # 1 'space' is 1 byte
                        # so this kind of allocation in particular can be used to all
```

Pointers

```
# set pointer
# *ecx = &x;

movl $x, %ecx


# dereference pointer
# ebx = *(ecx) or ebx = x[0]

movl (%ecx), %ebx

# pointer incrementation
# ecx = (ecx + 4) or ecx = x[1]

addl $4, %ecx
```

Advance Indexing

Globals

```
# eax is i
# ebx is j

movl a(, %eax, wordsize), %ecx #ecx = a[i]
movl (%ecx, %ebx, wordsize), %ecx #ecx = a[i][j]
```

Stack

```
  #ecx = i, #edx = j
movl a(%ebp), %ebx #ebx = a
movl (%ebx, %ecx, wordsize), %ebx #ebx = a[i]
movl (%ebx, %edx, wordsize), %ebx #ebx is a[i][j]


# short cur_max = nums[0];
movl nums(%ebp), %ecx #ecx = nums
movl (%ecx), %edx
movl %edx, cur_max(%ebp)

----

#main[*counter][i] = share[i]
#share[i]
movl share(%ebp), %ecx
movl (%ecx,%edi,wordsize), %ecx #ecx = share[i]

#main[*counter]
movl counter(%ebp), %esi #esi = &counter
movl (%esi), %esi #dereference position @ counter
movl main(%ebp), %eax # eax = **main
movl (%eax, %esi, wordsize), %eax #main[*counter]
movl %ecx, (%eax, %edi, wordsize)  #main[counter][i] = share[i]


----

#share[select] = items2[i];
movl items2(%ebp), %ebx
movl (%ebx, %edi,wordsize), %ecx #ecx = items2[i]
movl share(%ebp), %eax
movl select(%ebp), %esi
movl %ecx, (%eax,%esi,wordsize)
```

Leal

```
#edi = i
movl values(%ebp), %eax     #eax = values
leal (%eax,%edi,wordsize), %ebx #ebx has values[i]
addl (%ebx),%ecx #deference memory location, add to ecx

----

#elems + 1, where elems is a pointer
movl elems(%ebp), %eax
leal wordsize*1(%eax), %eax #eax = elems + 1, next location in mem


----

int counter = 0;
int *p_counter = &counter;
leal counter(%ebp) , %eax

...

#(*counter)++;
movl counter(%ebp), %esi      #esi = &counter
movl (%esi), %esi               #dereference position @ counter
incl %esi                         #counter++
movl counter(%ebp), %eax     #move orginal counter to open register
movl %esi, (%eax)             #restore back original position (we already incremented
```

While Loops

```
                        # while(str[len] != 0)
    while_not_end:
        cmpb $0, str(%ebx)  # str[len] compared with 0
        jz end_while        # if == 0, end
                            # do your stuff
        incl %ebx           # len++
        jmp while_not_end   # go back
    end_while:
```

For Loops

Globals

```
# for(int i =0; i < num_rows; i++)

movl $0, %eax # i = 0
inner_loop:
    cmpl num_rows, %eax
    jge end_outer_loop

    ...

    incl eax
    jmp inner_loop
end_inner_loop:
```

Stack

```
# for(int i =0; i < num_rows; i++)
movl $0, i(%ebp) # i = 0
inner_loop:
    movl i(%ebp), %edi
    cmpl num_rows(%ebp), %edi
    jae end_inner_loop

    ...

    incl i(%ebp) #i++
    jmp inner_loop
end_inner_loop:
```

Function Mimicing

```
# Method 1 (bad)

jmp function1
back:


...


function1:
    # do something
jmp back


# Method 2 (better)

returnMin:
    # do your stuff
    ret

call returnMin
# returns here
```

Comparison Logic

- Note you want to have the complement of your whatever operation you will be doing in C, because all the comparisons are against zero. So an boolean expression like i < num_rows, would become i - num_rows >= 0, using the jge operator.

Jumps

Instruction	Description	Comparison Type
jmp label	Unconditional jump	NA
jl label	Jump if less than 0	Signed
jg label	Jump if greater than 0	Signed
jle label	Jump if less than or equal 0	Signed
jge label	Jump if greater than or equal 0	Signed
jb label	Jump if below 0	Unsigned
ja label	Jump if above 0	Unsigned
jbe label	Jump if below or equal 0	Unsigned
jae label	Jump if above or equal 0	Unsigned
jz label	Jump if zero	Signed or Unsigned
jnz label	Jump if not zero	Signed or Unsigned
jc label	Jump if carry	NA
jnc label	Jump if no carry	NA
jo label	Jump if overflow	NA
jno label	Jump if no overflow	NA

Saving Registers

- When: Variable are become vulnerable to change and use, so save thier current state.

```
# save c and i
movl %eax, c(%ebp)
movl %ecx, i(%ebp)
```

Restoring Registers

- After use we can go back to our original values.

```
#restore c and i
movl c(%ebp), %eax
movl i(%ebp), %ecx
```

Malloc

```
    #int** c = (int**) malloc((num_rows_a * size(int*);
    movl num_rows_a(%ebp), %eax #eax = mat_rows_a
    shll $2, %eax #eax = num_rows_a * size(int*)
    push %eax
    call malloc
    addl $1*wordsize, %esp #clear argument
    #Save C
    movl %eax, c(%ebp)


    #for(int i = 0; i < num_rows_a; i++)
    #edi = i
    #ebx = c
    #eax = num_cols_b
    #c[i] = (int*) malloc(num_cols_b * size(int*))
    movl c(%ebp), %ebx
    movl $0, %edi # i = 0
    intialized_loop:
    movl num_cols_b(%ebp), %eax
    shll $2, %eax
    push %eax
    call malloc
    movl  %eax, (%ebx, %edi, wordsize)
    incl %edi
    cmp num_rows_a(%ebp), %edi
    jl intialized_loop

    addl $1*wordsize, %esp #clear eax from intialized loop
```

Inline

Boiler

```
__asm__(
  assembly code :
  outputs :
  inputs :
  clobbered
);

outputs, inputs, and clobbered are all optional

*/
```

Variation Favorites

```
int altAnyRegAdd(int a, int b){
    //again allow gcc/g++ to choose which register
    //to store our values in but this time we give nice names
    //to the values instead of having to use their positions
    __asm__(
    "addl %[b], %[sum]" :
    [sum] "+r" (a): //note that the names of the variables don't have to match the names
                                    //up in the assembly.
    [b] "r" (b) //but they can if you want them to
    );
    return a;
}
```

```
int anyWhereAdd(int a, int b){
    //this time we allow gcc to store our values
    //in any register it wants or memory
    __asm__(
    "addl %[b], %[sum]" :
    [sum] "+g" (a): //the g means a general purpose register or somewhere in memory
    [b] "g" (b):
    "cc"
    );
    return a;

}
```

DDD tips from the command console

- s = step
- b [line number, or label]
- r = run
- p/x &x; print the address of x
- quit = interupt debugging process
- clear 29 = removes breakpoint at line 29.

Display an entire array in memory:

- Data>>memory
  - Examine: number of elements
  - Base to be read in
  - Size (e.g. word = .long (4 bytes))
  - from: starting &address element
- Very helpful because these values will be changes you progress your way through the program.
- Or right click on element in ddd and press 'display 'element"

When writing asm

- super pro tip: store everything into raw memory, and apply to registers when need be. Makes things so much easier to write.

Display a 2d array from the stack

- First place your array into a register say %eax. Then you can do

```
p ((int**)$eax)[i][j]
```

- Where i and j are the coordinates of the element you are interested in. i and j can be constants or they can be registers.

SAVE SESSIONS IN DDD TO AVOID HAVING TO RESET UP AFTER EVERY COMPILATION.

Bitwise Operators

Great Reference

- A note that C/C++ used arithmetic shift on singed values, and logical shift on unsigned numbers. This is determined by the compiler, but is usually the case.

Converting to a Binary Scientific Floating Point Number

```
unsigned int float_int = *((unsigned int *)&f);
```

- Converts to 2's complement, and maintains the precision of the number, as well as the sign (interally, that is).

```
cout << sizeof(float_int); // 4
cout << sizeof(float_int) * 8 - 1 << endl; //31
```

- 1 byte = 8 bits. Float_int will then contain 31 bits (start counting from 0). We can use this information to determine the number of bits we are working with, so we don't commit any overflow.

```
unsigned int sign = float_int >> 31;
```

- Bit shift operators work on the bits themselves directly, which are expressed in binary. The reference above is of great help in explaining what they do.
  - A negative value will use arithmetic, and will preserve the sign of the most significant bit.
  - Logical shift is not sign preserving, and explicitly shifts 0's to the appropriate amount.

```
float f = 1;
bitset<32> floatBit (f);
cout << floatBit << endl; //00000000000000000000000000000001
```

- We can see the binary representation of the float this way.
- Which is to be expected.
- What happens when we do this?

```
float f = -1;
cout << floatBit << endl; //111111111....1
```

- Why is this the case?
  - The most significant bit represents whether the number is a negative number.
  - So like the arithmetic shift, any shift will maintain '1' as a bit.
- But wait! bitset<> coerses the number to an integer form. We can prove this: float f = 1.5f; will return the same solution we've seen first.
- To fix this, we pass in the pointer, because pointers cannot be coerced.

```
int i_f = 1;
bitset<32> floatBit(*(unsigned*)(&1_f)); // 0000....1
```

- And when we use

```
int i_f = 1;
bitset<32> floatBit(*(unsigned*)(&1_f)); // 00111111100000000000000000000000
00111111111000000000000000000000


f = -27;
bitset<32> floatBit(*(unsigned*)(&f)); // 11000001110110000000000000000000
```

- The first 8 bits represent the exponent.
- The rest of the bits (8-32) = 24 represent the mantissa.
- So when we use the bitwise operator:

```
sign = float_int >> 31; // 1
```

- We can note whether the number was positive or negative.
- 1 = -
- 0 = +

Reading Bit Sequences

```
exponent = float_int & 0x7f800000; //01111111100000000000000000000000
```

- Lets take a look at these operators.
  - & is bit wise and preforms the following

```
11000001110110000000000000000000
01111111100000000000000000000000 // equivalent representation of 0x7f800000
```

- Interpreted: If we and the binary rep, with 1, then what will be returned will be directly what is within that bit.
- In our case, what is going to be returned is the following:

```
//return
01000001100000000000000000000000
```

- Essentially guarentees that everything else is 0. This information becomes lost.
- Because we understand that the mantissa is represented by the next 23 bits, or that the total size of our unsigned integer is 32 bits, we can do this to shift things down and treat it as an actual binary number.

```
exponent >>= 23;
bitset<32> temp(*(unsigned*)(&exponent));
cout << temp << endl;
//00000000000000000000000010000011
```

operator	asm equivalent	description
&	AND	Bitwise AND
|	OR	Bitwise inclusive OR
^	XOR	Bitwise exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift bits left
>>	SHR	Shift bits right

Assembly

- Will be reviewing
  - Linux intel x86 assembly langauge.

Hex Editor:

- Used to examine a variety of files:
  - jpg, png, exe.
  - Digits represented in hexidecimal.
  - Attempts to represent the numerical values as text (not always successful).
  - Offset: location in file.
  - Can change and modify values, just like any other text editor, but on a lower level.

The Binary Numeric System

Binary Addition and Subtraction:

- 0 = low voltage
- 1 = high voltage
- Key: The amount of ideas we can represent are in binary.
  - k things are represented by 2^k combinations. E.g. 3 objects can be distinct among 8 combinations. Thus floor(log(k)) objects are represented from 2^k combinations.
  - 1, and 0 are called bits

Addition

- Same as with your conventional base 10 addition, except that 2 is the combination 10, just like when we reach 9 + 1 = 10.
- e.g. 11+1=100 base 2

Subtraction

- Same idea, but the concept of borrowing we take borrow from the first least significant 1, change it to a zero, and then the proceeding zero becomes 10. Borrowing from 10 gives us one (10 - 1) = 1, since 2 - 1 = 1.
- e.g. 100 - 1 = 011

x86 Architecture

Basic History

- Processors:
  - Read and write from memory
  - Tasks are interpreted line by line
- x86 = family of processors
  - Developed from embedded systems (single function computers e.g. routers, cars)
  - 16 bit support
  - Then 32 bit .. and then 64.
- Other companies other than intel produce .86 compatable processors.
  - linux, max and windows
- Old programs all the way from 1978 work on new processors. Backwards compatable.
  - New processors have more operation modes
  - Real mode (old) limited memory access 1MB.
    - Default mode for the current processor. The OS moves on to newer modes.
  - Protected mode (newer): more memory 1GB.
    - Helps manage multitasking in the OS.
      - Open more than one app.
  - Long mode:
    - Handles 64 bit data

Structure

- Programs written in RAM memory.
- byte = 8 bits
- hexadecimal digits or nibble = 4 bits
- e.g. 89 C1 01 C9 01 C1
  - These are characters represented in base 16. AKA Opcode.
  - They are interpreted as instructions.
  - 89 C1 = move ecx, eax
  - 01 C9 = add ecx, ecx
  - 01 C1 = add ecx,eax
- We either add, subtract, or move data
- Instructions sizes all vary, from 1 byte to 10 bytes
  - Can only read the numerical representation of the instructions.

    - We work with a textual representation.
- Operation Cycle:
    - read instruction from memory
    - decode the instruction (what it means)
    - execute
    - fetch the next instruction
- The registers:
    - where data is stored
    - built in processor, or CPU
    - very efficient
    - more efficient than RAM
    - very scarce
    - short term memory
    - represent the architecture of the CPU
- What the registers are (32 bits each)
    - eax - accumulator (adding)
    - ebc - base index
    - ecx - counter (forward for backwards)
    - edx - data register
    - These registers can do the operations of the other registers, but it is important to know that some registers are more efficeint than others.
    - e = 'extended'
- Register extensions:
    - backwords compatability

<table>
<tr><td colspan="3">Eax – 32 bit</td></tr>
<tr><td></td><td colspan="2">Ax – 16 bit</td></tr>
<tr><td></td><td>Ah – 8 bit</td><td>Al – 8 bit</td></tr>
</table>

- Registers can be broken down into subcomponents - for backwords compatability
    - ax is known as the lower 16 bits or eax
    - h = high
    - l = low
- e.g. 0xabcd1234
    - ax = 0x1234
    - ah = 0x12
    - al = 0x34
    - ecx = 0xabcd
- Each register has this same property. That is to say, ebx, ecx, and edx all have this same structural branching.
- 64 bit version:

    - Same thing, expect the subcomponents are branched reletive to the initial 64 bit size.
    - e.g. 64/32/16/8
    - long mode gives us access to this
- More registers: (32 bit)
    - esi = source index register
    - edi = destination index register
    - subcomponents are limited to 16 bits max (one subsystem)
- Instruction pointer = epi
- Flag registers
- Stack pointers
    - esp = stack pointer
    - ebp = base pointer
- Note that initially the registers have some garbage/unknown value inside them.

First Instructions

- we use mnemonic or shortcut for an instructions name (easier than numeric)
- structure:
    - mneumonic arg1, arg2...
    - arguements are encoded into the numeric representation (that actual assembly)
    - text is tranlated - aka assembler
    - the assembler is what interprets the textual format of assemby
        - various assemblers have different tastes in syntax
        - thats why we have different assemblers like fasm or flat, or nasm
- Instructions:
    - move = COPY data
        - mov destination, src
        - mov eax, 8CBh
            - h = read as base 16
            - so we copy the number to the eax 32 bit register
        - mov ecx, edx
            - copy edx to ecx register
        - mov si, cx
            - copy cx subregister (16 bit) to si subregister (16 bit)
        - operation vary in size
        - error:
            - mov 13h, ecx
            - because we are copying a 32 bit register to a number. A number is not a

destination.

- error:
  - move ecx, dh
  - because the components must also carry the same size
  - dh = 8 bits
  - ecx = 32 bits

instruction	eax	ecx	edx
	????????	????????	????????
mov eax, 3h	00000003	????????	????????
mov edx, ABh	00000003	????????	000000AB
move edx, edx	00000003	????????	000000AB
mov ecx, edx	00000003	000000AB	000000AB
mov edx, eax	00000003	000000AB	00000003

- Analysis:
  - start with some existing garbage value
  - copy and move to existing locations in the cpu
  - notice the overwrites
  - notice that the entire 32 bits of information are overwritten.

instruction	eax	ecx
	????????	????????
mov ax, 9Ch	????009C	????????
mov eax, DDDD1234	DDDD1234h	????????
move cl, E5h	DDDD1234	??????5E
move ah, cl	DDDD5E34	??????5E

- Analysis:
  - It is important to recognize that ax and 9Ch are both 16 bits, so they will take the lower half of ecx 32 bit information of the previous value.
  - Next, moving to eax completely overwrites the existing value, both consist of 32 bits.
  - cl is the lower half of cx, so it is 8 bits.
  - the last one is the most tricky: cl is taken, and move to another 8 bit location within another register. We know that it much be 8 bits becasue it much be the same size. ah, is part of ax (the higher portion), so it overwrites the previous ah, within eax.

First instructions part 2

- adding
  - ADD destination, src
  - like: destination+=source
  - ex:
    - add eax, edx
      - eax+=edx
    - add esi,11b
      - esi+=11b
      - b = converts to binary
      - esi+=3
    - add dx, si
      - dx+=si
      - 16 bit addition
      - result then wrap arrounds 16 bits
    - add 532h, ecx
      - error: we cannot add a literal (number) to a register
      - why: because we need to store it in some location.
    - add bx, eax
      - error: we cannot add a 32 bit number to a 16 bit number
      - why: where do we store the remaining bits?

instruction	esi	eax	ebx
	00000001	00000002	00000003
add eax, ebx	00000001	00000005	00000003
add eax, eax	00000001	0000000A	00000003
add esi, 0FFFFFFFFh	FFFFFFFF	0000000A	00000003
add ebx, esi	FFFFFFFF	0000000A	00000002
add esi, eax	00000009	0000000A	00000002

- Analysis: column by column
  - 2 += 3
  - 5 += 5; or A in hex
  - 1 += FFFFFFFF; = FFFFFFFF
    - So thoughts below
    - this works because we start from 0
  - 3 += FFFFFFFF; = FFFFFFFF
    - WHAT WHYYYYYYYYYYYYYY

        - Two things FFFFFFFF is a very large binary number, which consists of all 1's.
        - When we add 3 to this, we should expect some wrap around, because we would otherwise require more than 32 bits.
        - We start from 0 again. 0,1,2 giving us the result we now expect, 2
    - FFFFFFF += 0000000A
        - Same reasoning above, start from 0, and counter to 10.

instruction	edi	ecx	eax
	AB29FFFF	00000703	000000FF
add al, ch	AB29FFFF	00000703	00000006
add di, cx	AB290702	00000703	00000006
add edi, 0AB29FFFFh	AB29FFFF	00000703	00000006
add edi, ecx	AB2A0702	00000703	00000006

- Analysis:
    - add ch = cx higher 8 bits = 07 to al or the lower 8 bits of ax = FF
        - FF+=07 = wrap around = 06
        - pheeewh
    - di = 16 bits of edi, or FFFF+= to cx or second half of ecx = 0703
        - FFFF+=0703; another wrap around 0703 = 0703 - 1 = 0702
    - AB290702 += 0AB29FFFFh; = AB29FFFF
        - Hex addition is pretty easy when you know that F = max, so it overwrite everything
    - AB29FFFF+=00000703; = AB2A0702
- Wraparounds work based on the size of the arguements.

Subtraction

- sub destination, src
    - destination-=src
- -src is found using 2's complement.
- Examples
    - sub eax, edx
        - eax -=edx
    - sub cl,dl
        - cl-=dl (8 bit operation)
    - sub eax, dl
        - error: not same size
    - sub 1Ah, dl

- Like before, we need a place to store our information. A number does not suffice.

Instruction	Eax	Ebx	Ecx
	0000001A	00000003	00000002
Sub eax, ebx	00000017	00000003	00000002
Add eax, ebx	0000001A	00000003	00000002
Sub ecx, ebx	0000001A	00000003	FFFFFFFF
Add ecx, eax	0000001A	00000003	00000019
Sub cl, al	0000001A	00000003	000000FF

0000001A-00000003 = 26-3 = 20 which is 17

00000017+00000003 = 1A

00000002-00000003 = FFFFFFF

FFFFFFF+0000001A = 19

19-1A = FF

- Some things to note:
  - our representations are in hex, so our calculations are done in hex
  - In our final case, the wrap around only extends 8 bits - the maximum size of the subregister!!!

x86 Architecture pt.2

Basic Arithmetic:

- inc, dec - increase and decrease
  - by 1
  - inc destination
  - dec destination
  - wrap arounds exists on overflow or underflow, based of size of number
  - examples
    - inc eax
      - eax+=1
    - dec si
      - si -=1
    - inc 1C5h
      - error: a number is not a destination
      - 1C5+=1 seems to be able to work intuitively, but we need to be able to store the result. A literal is not a place.

Instruction	eax
	FFFFFFFE
inc eax	FFFFFFFF
inc al	FFFFFF00
dec al	FFFFFFFF
inc ax	FFFF0000
dec ax	FFFFFFFF
inc eax	00000000
inc eax	00000001

- Analysis:
  - FFFFFFFE += 1; = 11111111111111111111111111111110 + 1 = 11111111111111111111111111111111 or FFFF FFFF
  - al = lower FF += 1; overflow = FFFFFF00
  - al = lower 00 -= 1; underflow = FFFFFFFF
  - ax = FFFF+= 1; overflow = FFFF0000
  - ax = 0000-= 1; underflow = FFFFFFFF

    - eax = FFFFFFFF++; overflow = 00000000
    - eax = 00000000++; 00000001

- mul - multiplication
    - unsigned (only positive) numbers
    - syntax:
        - mul arguement
        - single arguement
    - Example:
        - ax arguement
    - Forms:
        - ax = al * argument (8 bits)
        - dx:ax = ax * argument (16 bits)
        - edx:eax = eax * argument (32 bit)
        - What this means:
            - the result stored is twice the size of the argument
                - why this is the case:
                    - It ensures no wrap around, and we get the result we precisely want.
                    - For example dx:ax = ax * argument
                        - both ax and argument are 16 bits. Regardless of what their values are, the maximum possible value will be 32 bits. There size cannot exceed the result.
            - : means bit concatenation
                - we can extended a subsystem with this operator.
                - dx:ax = 32 bits (16 bits each)
                - edx:eax = 64 bits
        - Example:
            - mul ecx
                - edx:eax = eax * ecx
            - mul si
                - dx:ax = ax * si
            - mul al
                - ax = al * al
            - mul 2Ah
                - error: 2A is technically an argument, but this opcode cannot exist.

Instruction	Edx	Eax	Ecx
	AB1E2FFF	00000003	00000002
Mul ecx	00000000	00000006	00000002
Mul ecx	00000000	0000000C	00000002
Mov ax, 0EEEEh	00000000	0000EEEE	00000002
Mul ax	0000DEFF	00006544	00000002
Mul cl	0000DEFF	00000088	00000002

1. Mult Ecx; form = edx:eax = eax * ecx ;
    a. Edx:eax = AB1E2FFF+00000003= AB1E2FFF 00000003
    b. 00000003*00000002=00000000 00000006
2. Mult Ecx; form = edx:eax = eax * ecx;
    a. Edx + eax = 00000000 + 00000006 = 00000000 00000006
    b. 00000006 * 00000002= 12 or 0000000C in hex
3. Mov destination, src
    a. Translation: copy EEEEh to the lower end of ax
    b. 000C = EEEE
4. Mul ax;
    a. Ax contains the form dx:ax = Ax * ax (32 bit)
        i. EEEE * EEEE = DEFF 6544
    b. Notice only the first 16 bits are manipulated respectively.
5. Mul cl
    a. Contains the form ax = al * cl (8 bits each)
    b. 44 * 02 = 0088

- Things to be kept in mind:
    - syntax is from fasm perspective
    - GNU version can be referenced blow

Multiplication and Division

Operation	Result	Type	Affects Flag Register?
mulb src	ax = al * src	unsgined	Yes
mulw src	dx:ax = ax * src	unsgined	Yes
mull src	edx:eax = eax * src	unsgined	Yes

- div - division

First Program

- gas will be the main assembler I will use.
- Lets write our first program. I will be using asm file called temp.s for practice, along with a makefile for convienience.
- The Makefile:

```
temp.out: temp.o
    ld -melf_i386 -o temp.out temp.o
temp.o: temp.s
    as --32 --gstabs -o temp.o temp.s
clean:
    rm -f temp.o temp.out
```

- And then run through it with ddd:

```
ddd temp.out
```

- We make sure to break at the label 'done' to make sure our program does not seg fault.

Tips in writing in asm:

- Use the top down approach
    - Begin operation definitions, and then specify downwards.
    - Begins with main() e.g. _start
        - Should be very small. In general, limit each function to about a dozon or so lines of code.
- The first program I will take a look at will sum integers within an array.
- To see the result, we break at done - and view the value of the registers.
- The general process of writing asm is to first wright C, and then translate the logic into asm.

```
# introductory example; finds the sum of the elements of an array

.data # start of data section

x:
    .long 1
    .long 5
    .long 2
    .long 18

sum:
    .long 0

.text # start of code section

.globl _start
_start:
    movl $4, %eax # EAX will serve as a counter for
    # the number of words left to be summed
    movl $0, %ebx # EBX will store the sum
    movl $x, %ecx # ECX will point to the current
    # element to be summed
top: addl (%ecx), %ebx
    addl $4, %ecx # move pointer to next element
    decl %eax # decrement counter
    jnz top # if counter not 0, then loop again
done: movl %ebx, sum # done, store result in "sum"
```

Analysis

- C style code will be reference before the asm for clarity when appropriate.

```
// this is a comment

# introductory example; finds the sum of the elements of an array
```

- # Comment

```
.data # start of data section
```

- .data
  - First component of asm file.
  - '.' means directive - informing the compiler to that the following section will contain data, or variables.

```
int x[4] = {1,5,2,18};

x:
    .long 1
    .long 5
    .long 2
    .long 18
```

- labels:
  - x:
    - naming convention for your own reference.
    - Whatever is referenced below this label will be represented by this label.
    - This entire long sequence of memory will be referenced by 'x'.
    - We will typically use labels if we want to reference to them later in the program.
    - Note that labels are only useful conventions for humans. They are NOT actually variables, but it can be helpful to interpret them this way.
- .long:
  - Storage in memory.
  - long means 32 bit size, so whatever follows long is the actual 32 bit number.
  - Can be placed consecutively and will develope a consecutive blocks of memory.
  - This effectively creates an array.
  - Protip: what if we wanted to represent an array of 1000 elements?
    - .rept
    - Example:

```
int x[1000];
for (int i =0; i< 1000;i++) x[i] = 8;

x:
    .rept 1000
    .long 8
    .endr
```

- Anything between .rept and .endr will repeated inline 1000 times.
- For something of smaller size, we can use .space to preform something similiar.

```
char y[6];

y:
    .space 6
```

- Reserves 6 BYTES of space under the label y.
- Or if we wish to immediately initialize a space with some value, we can do this:

```
char y[] = "hello";

y:
    .string "hello"
```

- Which allocated 6 bytes (one for every character, including the null char), which is all preformed automatically with the .string directive.

```
sum = 0;

sum:
    .long 0
```

- Same thing as before, but because we want to represent the sum an independent variable, we set it under its own label.

```
.text # start of code section
```

- .text is the second component of asm, which signals that anything that comes after it will be part of the actual code.

```
.globl _start
_start:
```

- .global puts _start under the global namespace, so your linker knows where to begin.
- _start is a label that signals where your program actually begins execution.

```
    // what we are going to accomplish next:
    for (int i =0; i -4 <= 0 ; i--)
        sum+=x[i];


    eax = 4;

    movl $4, %eax # EAX will serve as a counter for
                  # the number of words left to be summed
      movl $0, %ebx # EBX will store the sum
```

- movl will copies the value 4, and places it into the eax register.
- $denotes constant values.
  - Otherwise, it would denote an address.
  - 'l' denotes 'long' or 32 bit, which also for the appropriate size of the register.
- Note the registers used are generally arbitrary, but they are used to operate arthmetic

and memory operations under the hood in the CPU.
- It is very important to use the registers as much as we possibly can, and especially when doing repeatative for loops. If we were to instead directly work with variable stored in memory, it would be significantly slower. Using registers utalizes cache within the CPU directly, and it operates much quicker.
- CPUs do however, have a limited number of registers - so it may be sometimes nessessary to use some outside memory from RAM.

```
    *ecx = &x;

    movl $x, %ecx # ECX will point to the current
                  # element to be summed
```

- $ here still denotes a constant like before, except the constant is the address of x. ecx then effectively becomes a pointer.

```
ebx = ebx + (ecx)

top:
    addl (%ecx), %ebx
```

- The next line adds whatever was stored within the address of ecx into ebx (our main storage register). Ecx previously pointed to the address of x, which was the first location in our array x.
- Addl will take this 32 bit register and storage it into another register.

```
ecx = (ecx + 4)

addl $4, %ecx # move pointer to next element
```

- ecx holds the address of our array, but as a pointer. Therefore, when we increment this pointer, we are mimicing pointer arithmetic, and move to the next location of 32 bits. 4 means move 4 bytes (in memory), which translates to 4*8=32 bits. Therefore, ecx will then point to the start of the next 'number' in the consecutive memory block.

```
eax--;

decl %eax # decrement counter

jnz top # if counter not 0, then loop again
```

- eax is our original counter in our for loop, initially beginning at 4.

- The next operation, jnz together integrates with the operation above it.
  - This is the EFLAGS register
  - Within an arithmetic instruction in the CPU, it records whether the result of the instruction is 0. When it is, it sets the zero flag in the EFLAGS register to 1, to signal that the result of the operation was true.
  - jnz can be interpreted as the command 'jump if not zero', which says 'hey, if the result of your last operation was not zero, then go back to 'top'.
  - There also the complementary incruction 'JZ' which says 'if the last operation was zero, then go to _.
- Together with our decrement operation, we will be able to affectively loop 4 times (the size of the array), and store our sum within the register.

```
    sum = ebx;

    done: movl %ebx, sum # done, store result in "sum"
```

- This final label is not required, or prehap redundent because we dont reference it nowhere in the program, but its a design choice.
- Our final operation stores our result into memory label, sum.

Inline Assembly

//source: http://asm.sourceforge.net/articles/rmiyagi-inline-asm.txt

```
-----------------------------------------------------------------------
             Introduction to GCC Inline Asm

                By Robin Miyagi

       http://www.geocities.com/SiliconValley/Ridge/2544/

            Wed Sep 13 19:18:50 UTC
-----------------------------------------------------------------------

* `as' and AT&T Syntax
-----------------------------------------------------------------------

    The  GNU C  Compiler uses  the assembler  `as' as  a  backend.  This
    assembler uses AT&T syntax.  Here is a brief overview of the syntax.
    For  more   information  about  `as',   look  in  the   system  info
    documentation.

    - as uses the form;

        nemonic source, destination (opposite to intel syntax)

    - as  prefixes registers  with `%',  and prefixes  numeric constants
      with `$'.

    - Effective addresses use the following general syntax;

        SECTION:DISP(BASE, INDEX, SCALE)

      As in other assemblers, any one or more of these components may be
      ommited,  within constraints  of valid  intel  instruction syntax.
      The above syntax was shamelessly  copied from the info pages under
      the i386 dependant features of as.

    - As suffixes  the assembler nemonics  with a letter  indicating the
      operand sizes  ('b' for  byte, 'w' for  word, 'l' for  long word).
      Read  the info  pages for  more information  such as  suffixes for
      floating point registers etc.

    Example code (raw asm, not gcc inline)
    ---------------------------------------------------------------------
    movl %eax, %ebx     /* intel: mov ebx, eax */
    movl $56, %esi      /* intel: mov esi, 56 */
    movl %ecx, $label(%edx,%ebx,$4) /* intel: mov [edx+ebx*4+4], ecx */
    movb %ah, (%ebx)    /* intel: mov [ebx], ah */
```

```
    ---------------------------------------------------------------------

    Notice that  as uses  C comment  syntax.  As can  also use  `#' that
    works the same way as `;' in most other intel assemblers.

    Above code in inline asm
    ---------------------------------------------------------------------
      __asm__ ("movl %eax, %ebx\n\t"
           "movl $56, %esi\n\t"
           "movl %ecx, $label(%edx,%ebx,$4)\n\t"
           "movb %ah, (%ebx)");
    ---------------------------------------------------------------------

    Notice that in the above example, the __ prefixing and suffixing asm
    are not neccesary,  but may prevent name conflicts  in your program.
    You can read  more about this in [C  enxtensions|extended asm] under
    the info documentation  for gcc.

    Also notice the '\n\t' at the  end of each line except the last, and
    that each  line is  inclosed in quotes.   This is because  gcc sends
    each as instruction to as  as a string.  The newline/tab combination
    is required so that the lines are fed to as according to the correct
    format  (recall that  each line  in asssembler  is indented  one tab
    stop, generally 8 characters).

    You can also use labels from  your C code (variable names and such).
    In  Linux, underscores prefixing  C variables  are not  Necessary in
    your code; e.g.

       int main (void) {
           int Cvariable;
           __asm__ ("movl Cvariable, %eax"); # Cvariable contents > eax
       __asm__ ("movl $Cvariable, %ebx"); # ebx ---> Cvariable
       }

     Notice that  in the documentation for  DJGPP, it will  say that the
     underscore is  necessary.  The difference is do  to the differences
     between  djgpp RDOFF  format  and  Linux's ELF  format.   I am  not
     certain, but I think that the old Linux a.out object files also use
     underscores (please contact me if you have comments on this).

* Extended Asm
-------------------------------------------------------------------------

    The code  in the  above example will  most probably  cause conflicts
    with the rest of your C code, especially with compiler optimizations
    (recall that gcc is an  optimizing compiler).  Any registers used in
    your code may be used to hold  C variable data from the rest of your
    program.  You  would not want  to inadvertently modify  the register
    without telling gcc to take  this into account when compiling.  This
    is where extended asm comes into play.

    Extended  asm   allows  you  to  specify   input  registers,  output
```

registers, and clobbered registers as interface information to your block of asm code. You can even allow gcc to choose actual physical CPU registers automatically, that probably fit into gcc's optimization scheme better. An example will demonstrate extended asm better.

Example code

```
#include <stdlib.h>

int main (void) {
  int operand1, operand2, sum, accumulator;

  operand1 = rand (); operand2 = rand ();

  __asm__ ("movl %1, %0\n\t"
           "addl %2, %0"
       : "=r" (sum)            /* output operands */
       : "r" (operand1), "r" (operand2) /* input operands */
       : "0");                /* clobbered operands */

  accumulator = sum;

  __asm__ ("addl %1, %0\n\t"
       "addl %2, %0"
       : "=r" (accumulator)
       : "0" (accumulator), "g" (operand1), "r" (operand2)
       : "0");
  return accumulator;
}
```

The first the line that begins with ':' specifies the output operands, the second indicates the input operands, and the last indicates the clobbered operands. the "r", "g", and "0" are examples of constraints. Output constraints must be prefixed with an '=', as in "=r" (= is a constraint modifier, indicating write only). Input and output constraints must have its correspoding C argument included with it enclosed in parenthisis (this must not be done with the clobbered line, I figured this out after an hour of fustration). "r" means assign a general register register for the argument, "g" means to assign any register, memory or immediate integer for this.

Notice the use of "0", "1", "2" etc. These are used to ensure that when the same variable is indicated in more than one place in the extended asm, that is variable is only `mapped' to one register. If you had merely used another "r" for example, the compiler may or may not assign this variable to the same register as before. You can surmise from this that "0" refers to the first register assigned to a variable, "1" the second etc. When these registers are used in the asm code, they are refered to as "%0", "%1" etc.

```
Summary of  constraints. (copied from the  system info documentation
for gcc)
---------------------------------------------------------------------
`m'

    A memory operand  is allowed, with any kind  of address that the
    machine supports in general.

`o'

    A  memory  operand  is  allowed,  but only  if  the  address  is
"offsettable".    This  means  that   adding  a   small  integer
(actually, the width  in bytes of the operand,  as determined by
its machine mode) may be added  to the address and the result is
also a valid memory address.

For example, an address which  is constant is offsettable; so is
an address that is the sum of a register and a constant (as long
as  a slightly  larger  constant  is also  within  the range  of
address-offsets supported by  the machine); but an autoincrement
or autodecrement  address is not  offsettable.  More complicated
indirect/indexed  addresses  may   or  may  not  be  offsettable
depending  on  the  other  addressing  modes  that  the  machine
supports.

Note that in  an output operand which can  be matched by another
operand,  the   constraint  letter   `o'  is  valid   only  when
accompanied by both `<'  (if the target machine has predecrement
addressing)  and `>'  (if  the target  machine has  preincrement
addressing).

`V'

    A  memory operand  that  is not  offsettable.   In other  words,
anything  that would  fit the  `m'  constraint but  not the  `o'
constraint.

`<'

    A   memory   operand   with   autodecrement   addressing   (either
predecrement or postdecrement) is allowed.

`>'

    A   memory   operand   with   autoincrement   addressing   (either
preincrement or postincrement) is allowed.

`r'

    A register operand  is allowed provided that it  is in a general
register.

`d', `a', `f', ...
```

Other letters can be defined in machine-dependent fashion to stand for particular classes of registers. `d', `a' and `f' are defined on the 68000/68020 to stand for data, address and floating point registers.

`i'

An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time.

`n'

An immediate integer operand with a known numeric value is allowed. Many systems cannot support assembly-time constants for operands less than a word wide. Constraints for these operands should use `n' rather than `i'.

`I', `J', `K', ... `P'

Other letters in the range `I' through `P' may be defined in a machine-dependent fashion to permit immediate integer operands with explicit integer values in specified ranges. For example, on the 68000, `I' is defined to stand for the range of values 1 to 8. This is the range permitted as a shift count in the shift instructions.

`E'

An immediate floating operand (expression code `const_double') is allowed, but only if the target floating point format is the same as that of the host machine (on which the compiler is running).

`F'

An immediate floating operand (expression code `const_double') is allowed.

`G', `H'

`G' and `H' may be defined in a machine-dependent fashion to permit immediate floating operands in particular ranges of values.

`s'

An immediate integer operand whose value is not an explicit integer is allowed.

This might appear strange; if an insn allows a constant operand with a value not known at compile time, it certainly must allow

any known value. So why use `s' instead of `i'? Sometimes it allows better code to be generated.

For example, on the 68000 in a fullword instruction it is possible to use an immediate operand; but if the immediate value is between -128 and 127, better code results from loading the value into a register and using the register. This is because the load into the register can be done with a `moveq' instruction. We arrange for this to happen by defining the letter `K' to mean "any integer outside the range -128 to 127", and then specifying `Ks' in the operand constraints.

`g'

Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.

`X'

Any operand whatsoever is allowed, even if it does not satisfy `general_operand'. This is normally used in the constraint of a `match_scratch' when certain alternatives will not actually require a scratch register.

`0', `1', `2', ... `9'

An operand that matches the specified operand number is allowed. If a digit is used together with letters within the same alternative, the digit should come last.

This is called a "matching constraint" and what it really means is that the assembler has only a single operand that fills two roles considered separate in the RTL insn. For example, an add insn has two input operands and one output operand in the RTL, but on most CISC machines an add instruction really has only two operands, one of them an input-output operand:

```
addl #35,r12
```

Matching constraints are used in these circumstances. More precisely, the two operands that match must include one input-only operand and one output-only operand. Moreover, the digit must be a smaller number than the number of the operand that uses it in the constraint.

For operands to match in a particular case usually means that they are identical-looking RTL expressions. But in a few special cases specific kinds of dissimilarity are allowed. For example, `*x' as an input operand will match `*x++' as an output operand. For proper results in such cases, the output template should always use the output-operand's number when printing the operand.

```
`p'

An operand that  is a valid memory address  is allowed.  This is
for "load address" and "push address" instructions.

`p' in  the constraint must be  accompanied by `address_operand'
as  the  predicate   in  the  `match_operand'.   This  predicate
interprets the mode specified in the `match_operand' as the mode
of the memory reference for which the address would be valid.

`Q', `R', `S', ... `U'

Letters  in  the range  `Q'  through `U'  may  be  defined in  a
machine-dependent fashion to  stand for arbitrary operand types.
The machine  description macro `EXTRA_CONSTRAINT'  is passed the
operand as its  first argument and the constraint  letter as its
second operand.

A typical use for this  would be to distinguish certain types of
memory references that affect other insn operands.

Do  not  define  these  constraint letters  to  accept  register
references  (`reg'); the reload  pass does  not expect  this and
would not handle it properly.

    In order to have valid assembler code, each operand must satisfy
its constraint.   But a  failure to do  so does not  prevent the
pattern  from applying  to  an insn.   Instead,  it directs  the
compiler  to modify  the code  so  that the  constraint will  be
satisfied.  Usually  this is done  by copying an operand  into a
register.

    Contrast, therefore, the two instruction patterns that follow:

 (define_insn ""
   [(set (match_operand:SI 0 "general_operand" "=r")
     (plus:SI (match_dup 0)
          (match_operand:SI 1 "general_operand" "r")))]
   ""
   "...")

which has two operands, one  of which must appear in two places,
and

 (define_insn ""
   [(set (match_operand:SI 0 "general_operand" "=r")
     (plus:SI (match_operand:SI 1 "general_operand" "0")
          (match_operand:SI 2 "general_operand" "r")))]
   ""
   "...")

which  has  three operands,  two  of  which  are required  by  a
constraint to  be identical.  If  we are considering an  insn of
```

```
the form

 (insn N PREV NEXT
   (set (reg:SI 3)
    (plus:SI (reg:SI 6) (reg:SI 109)))
   ...)

the first pattern would not apply at all, because this insn does
not  contain two  identical subexpressions  in the  right place.
The  pattern  would  say,  "That  does  not  look  like  an  add
instruction; try other patterns."  The second pattern would say,
"Yes, that's  an add instruction,  but there is  something wrong
with it."   It would direct the  reload pass of  the compiler to
generate  additional insns  to  make the  constraint true.   The
results might look like this:

 (insn N2 PREV N
   (set (reg:SI 3) (reg:SI 6))
   ...)

 (insn N N2 NEXT
   (set (reg:SI 3)
    (plus:SI (reg:SI 3) (reg:SI 109)))
   ...)

It is up to you to make sure that each operand, in each pattern,
has constraints that can handle any RTL expression that could be
present for  that operand.   (When multiple alternatives  are in
use, each pattern must, for each possible combination of operand
expressions, have at least one alternative which can handle that
combination of operands.)  The constraints don't need to *allow*
any  possible  operand--when  this  is  the case,  they  do  not
constrain--but they must at least point the way to reloading any
possible operand so that it will fit.

    * If  the  constraint accepts  whatever  operands the  predicate
  permits, there is no problem: reloading is never necessary for
  this operand.

  For  example, an operand  whose constraints  permit everything
  except  registers  is  safe  provided  its  predicate  rejects
  registers.

  An  operand whose  predicate accepts  only constant  values is
  safe provided its constraints  include the letter `i'.  If any
  possible constant  value is  accepted, then nothing  less than
  `i'  will do;  if the  predicate is  more selective,  then the
  constraints may also be more selective.

    * Any operand  expression can be  reloaded by copying it  into a
  register.  So  if an operand's constraints allow  some kind of
  register, it  is certain to be  safe.  It need  not permit all
  classes  of  registers;  the  compiler  knows how  to  copy  a
```

```
      register into another register of the proper class in order to
      make an instruction valid.

        * A nonoffsettable  memory reference can be  reloaded by copying
      the address  into a register.   So if the constraint  uses the
      letter `o', all memory references are taken care of.

        * A  constant operand  can be  reloaded by  allocating  space in
      memory  to hold it  as preinitialized  data.  Then  the memory
      reference can  be used  in place of  the constant.  So  if the
      constraint uses the letters  `o' or `m', constant operands are
      not a problem.

        * If  the constraint permits  a constant  and a  pseudo register
      used in  an insn was not  allocated to a hard  register and is
      equivalent to  a constant, the register will  be replaced with
      the constant.  If the predicate does not permit a constant and
      the insn  is re-recognized for some reason,  the compiler will
      crash.  Thus  the predicate must always  recognize any objects
      allowed by the constraint.

    If  the operand's  predicate  can recognize  registers, but  the
    constraint does not permit them, it can make the compiler crash.
    When this operand happens to be a register, the reload pass will
    be  stymied, because it  does not  know how  to copy  a register
    temporarily into memory.

    If  the  predicate  accepts  a unary  operator,  the  constraint
    applies to the operand.  For  example, the MIPS processor at ISA
    level  3 supports  an instruction  which adds  two  registers in
    `SImode' to produce a `DImode' result, but only if the registers
    are  correctly  sign extended.   This  predicate  for the  input
    operands accepts a `sign_extend' of an `SImode' register.  Write
    the constraint to indicate the type of register that is required
    for the operand of the `sign_extend'.
    ---------------------------------------------------------------------------

    The '='  in the  "=r" is  a constraint modifier,  you can  find more
    information  about  constraint  modifiers,  in the  gcc  info  under
    Machine Descriptions : Constraints : Modifiers.

    I strongly recommend reading  more in the system info documentation.
    If  you haven't  had  much  experience with  the  info reader  (also
    accesable through  emacs), learn  it, it is  an excellent  source of
    information.

    The gcc info  documentation also explains how to  use a specific CPU
    register for  a constraint for various hardware  including the i386.
    You  can  find  this  information   under  [gcc  :  Machine  Desc  :
    Constraints : Machine Constraints] in the info documentation.

    You can specify specific registers in your constraints, e.g. "%eax".
```

```
* __asm__ __volatile__
--------------------------------------------------------------------------

    Because of  the compilers optimization mechanism, your  code may not
    appear at  exactly in the  location specified by the  programmer.  I
    may  even be interspersed  with the  rest of  the code.   To prevent
    this, you can  use __asm__ __volotile__ instead.  Like  the '__' for
    asm, these  are also not needed  for volatile, but  can prevent name
    conflicts.

==========================================================================
comments and suggestions <deltak@telus.net>
```

My Notes

- Integrating into C, inline

```
__asm__(
)
or
asm(
)
```

- Although __ can help with naming conflicts
- Can directly take in C variables

Extended ASM

- Allows us to specify input registers, output registers, and clobbered registers - so that we don't inadvertently modify other registers.
- Indicated by collins:

```
    __asm__ ("movl %1, %0\n\t"
               "addl %2, %0"
         : "=r" (sum)            /* output operands */
         : "r" (operand1), "r" (operand2) /* input operands */
         : "0");                 /* clobbered operands */
```

  - 'r', 'g' and '0' are all constraints
  - input and output constraints must have their corresponding C arguement unclosed in paranthesis, while the clobber section cannot.
  - r = means to let the compiler assign the general register for you
  - g = same as r, but for any register.

    - = means write only (output). Can used to overwrite C variable.
    - "0", "1", "2" etc are used to for designation for the assigned registers. Because we cannot completely know what registers were used, 0, indicates the first, 1 the second, etc. But again, these are optional, and there are different ways to assign things.
        - So that way, we can use %0, %1, etc to indicate what registers we will use.
    -
        - means input and output
- Personally, I found these two to be my favorite variation:

```
int anyWhereAdd(int a, int b){
    //this time we allow gcc to store our values
    //in any register it wants or memory
    __asm__(
    "addl %[b], %[sum]" :
    [sum] "+g" (a): //the g means a general purpose register or somewhere in memory
    [b] "g" (b):
    "cc"
    );
    return a;

}
```

```
int altAnyRegAdd(int a, int b){
    //again allow gcc/g++ to choose which register
    //to store our values in but this time we give nice names
    //to the values instead of having to use their positions
    __asm__(
    "addl %[b], %[sum]" :
    [sum] "+r" (a): //note that the names of the variables don't have to match the names
                                  //up in the assembly.
    [b] "r" (b) //but they can if you want them to
    );
    return a;
}
```

Midterm1 Review

1. Everything in the computer is represented using
    i. bits
2. There are no (?) at assembly level.
    i. types
3. If you have N unique states, what is minimum number of bits needed to represent all of them?
    i. num(bits) = ceiling(log_2(states))
4. If you have B bits how many unique states can you represent?
    i. num(states) = 2^(bits)
5. How does typing in a high level language affect what assembly code will be generated?
    i. Higher level languages will generally have more restrictions that assembly. For example, it is possible to add two different variable types in assembly.
6. Be able to convert from one base to any other base:
    i. What is 456_7 in base 4?
        i. 3231
            i. convert to base 10, then base n
    ii. What is 01 1101 1111 in base 16?
        i. 1DF
            i. Sum individual sections of 4, using A-F for values larger than 16
    iii. What is 0xF57AC in binary?
        i. 11110101011110101100
            i. Represent each individual of hex as segments of 4.
7. Know the 2 different ways of representing singed numbers in binary
    i. Signed magnitude and 2's complement
    ii. For the following 8 bit numbers what are their values in Sign Magnitude and 2's compliment?
        i. 0010 1010
            i. Unsigned magnitude: 42
                i. Represent binary in decimal form.
            ii. Signed magnitude: 42
                i. Represent binary not including the most significant bit which will be used to represent +|-.
            iii. 2's complement: 11010101 + 1 = 11010110 = 214
                i. Reverse all the bits and add 1. Then represent the binary entirely in

decimal. Finally, use the most significant bit to represent the sign.

   ii. 1101 1110
      i. Unsigned: 222
      ii. Signed: -94
      iii. 2's complement: 00100001 + 1 = 100010 = -34

8. Know how a real number is stored using IEEE floating point format.
   i. [sign (1) bit] [exponent (8 bits)] [mantissa (23 bits)]
   ii. sign: 1 = -; 0 = +
   iii. exponent = exponent - 127
   iv. mantissa = the trailing zeros are trimmed off
   v. add "1." in the beginning
   vi. finaly form: "[sign]1.[mantissa]E[exponent]"
9. How is 57.25 stored in IEEE floating point format?
   i. 0) Note the sign = 0 (+)
   ii. 1) Convert float to binary form using C++ reinterpreted cast
      i. 111001.01
   iii. 3) Set to standard form: 1.1100101
   iv. 4) exponent = 127 + 5 = 132
   v. 5) convert exponent to binary form: 10000100
   vi. 6) Final answer: 0 1.1100101 10000100...(0*16)
10. Know the difference between little endian and big endian. Why is endianess an issue?
   i. Endianess determines the order in which something is stored into memory. It is an issue because different computers systems have different preferences for their endianess. So when these system try to communicate with one another, there can be a conflict on how to read digits from machine to machine.
      i. Big Endian: the 'big' or most significant bit is stored at the lowest memory address.
      ii. For example, if our name was 0x87654321, then we treat 87 with the lowest memory address, and assign the remaining bits in ascending address order.
      iii. Little endian is the opposite.

Byte	87	65	43	21
Address	0	1	2	3

1. If the value 0x 45 89 55 67 is stored in bytes 1000 – 1003 what is the value of the number being stored if the machine is little endian? Big endian?
2. Big endian: (read as 45895567)

Byte	45	89	55	67
Address	1000	1001	1002	1003

1. Little endian:(read as 67558945)

Byte	45	89	55	67
Address	1003	1002	1001	1000

1. What does byte addressable mean?
    i. In reality, instructions are are only byte addressable, we're data can be be accessed 8 bits at a time.
    ii. Every instruction is stored in an individual byte. For example, address 1002 containing a single instruction.
    iii. This is in contrast to 'word addressable' where an address in memory refers to, for example, a sequence of 3 bytes or instructions.
2. What are the two options for storing multidimensional arrays.
    i. 1) Array of arrays [][]
        i. Pros:
            i. Conceptually easier
            ii. Allows of raggedness (arrays can vary in size)
        ii. Cons:
            i. Pointer chasing: less efficient.
    ii. 2) Single arrays
        i. Pros:
            i. Faster (we just do a little bit more math)
        ii. Cons:
            i. We must know the dimension of the array, with exception to the last one, in order for this to work.
3. Given that Ar is a 3 dimensional array declared as Ar[2][3][4]. What is the single dimensional index of Ar[1][2][2] if Ar is stored in row major. In column major?
    i. The single dimension equivalent of getting Ar[d1][d2][d3] is:
        i. i *d2* d3 + j * d3 + k
        ii. 1 *3* 4 + 2 * 4 + 2 = 22
    ii. Column Major
        i. k *d1* d2 + j * d1 + i
        ii. 2 *2* 3 + 2 * 2 + 1 = 17
4. What does the following binary string represent 0110 1101?
    i. Because it is some binary string, we cannot be entirely sure without more context to

its representation.
    ii. We know for sure that it is something contained with 8 bits of information, but this could very well be a float, int, 2's complement int, signed or unsigned ... etc.
5. What are the 4 major steps executed by the CPU?
    i. 1) Fetch - Read information from memory, increment memory program counter.
    ii. 2) Decode - Interpret the information using register units
    iii. 3) Execute - Do it
    iv. 4) Store - Store the result in memory/register
6. What are the 2 major components that make up your computer?
    i. Hardware: CPU, RAM, input/output devices
    ii. OS: low level software that controls who get access to whose memory. Memory allocation, and maintain a file system (B+ tree)
7. Name at least 2 reasons why a program compiled on one computer may not run on another computer.
    i. 1) Because of different assemblers. While the general concepts are the same, there are variations in syntactical taste toward how assemblers read information. More importantly, different operating systems have different function calls, that every program relies on using and communicating with.
    ii. 2) Different hardware. The way the CPU structures its architecture can vary by machine. For example, the intel x86 processor vs the MIPS CPU.
8. Be able to correctly use the bitwise operators and bit shifting.
    i.
    ii. Construct a mask that once anded with a 32 bit number would allow you to examine bits 11, 5, and 3.
        i. In order to examine individual bits, our and operation must zero out all bits except for 11, 5, and 3. To do with we and all the binary elements with 0 except for bit numbers 11, 5, and 3. It should look something like this:
        ii. & 0000 0000 0000 0000 0000 0100 0001 0100
9. Negative numbers in hex:
    i. Convert original hex to 2's complement (binary, invert, and add 1), the convert the solution back to hex.

ECS 50 Midterm Review Guide

Know what all of the instructions on the Intel Cheat Sheet do and how to use them.

Understand how the location of a variable's declaration affects where it will be stored. Ie if a variable is local, global, or static where will space be made for it.

- Code is made up from four different parts:
- Code: or .text. This is where the body of instructions your code contains.
- Data: or .data. Where both global and static variables are contained.
- Stack: holds local variables, arguments from functions, and their return addresses.
- Heap: dynamic memory (i.e. with new in C++ or malloc).

Understand how typing in C affects the assembly instructions that are generated by the compiler.

- Each though no types can exist at assembly level code, this does not mean that different types assigned at higher level code will have same kind of instructions. Remember, the context of the problem will determine the kinds of instructions we will use. Pointers, for example, regardless if they are assigned to an int or char, will be 4 bytes of 32 bits. In addition any kind of arithmetic with pointers will be interpreted as pointer arithmetic.

Translate the following code to assembly. Assume that we want to store Assume that we want to store x in eax and c in ecx.

```
1 int* x = 100;
2 char* c = 70;
3 x += 10;
4 c += 10;


Translated
1 movl $100, %eax # store the memory address 100 into register
2 movl $70, %ecx # store memory address of 70 to register. We still use 'l' since all
3 addl $10*wordsize, %eax # here we're doing pointer arithmetic, so moving 10 locatio
4 addl $10*wordsize, %ecx # move a total of 10 bytes, since wordsize is 1 byte in thi
```

What are the gcc C calling conventions? What happens if a function breaks the calling conventions?

- 1) EAX, ECX, and EDX should not have live values when a function is called. This is because the function will use this registers, and possibly change the value in the

process. So, before calling a function, it is recommended save these values back onto the stack. This responsibility is placed on the caller of the function. It is highly encouraged to follow these conventions, even though you don't really have to. It is best to follow rules for consistency a set of guidelines for bug free results.

- 2) Functions will return into the EAX register. Again, you have control over this, but the general convention with gcc compiles are through this methods, so its better have this precaution.
- 3) Before calling a function, the arguments are pushed onto the stack, from right to left.

Understand and be able to use the advanced indexing mode. Assume that eax = 100 and ecx = 5. Which addresses in memory are accessed by the following instructions?

```
* general formula: displacement(base, index, scale); giving us the address at displacemen
* Restrictions
    * displacement = constant
    * base and index = registers
    * scale is a number from the set {1,2,4,8}.
    * default values respectively (if omitted): 0,0,0,1


    #eax = 100
    #ecx = 5
    1. movl (%eax), %ebx
    2. movb (%ecx,%eax), %bl
    3. movl %eax, %ebx
    4. movw (%ecx, %eax, 4), %bx


    1. 100 - 103 // eax is dereferenced, and 'l' indicates 4 bit locations.
    2. 105 // single bit 'b' at location 0+100+5*1 = 105
    3. Nothing is accessed. Only the addresses are moved around (not dereferenced)
    4. // size(w) = 3; at 0+5+100*4=405-408
```

What is considered the stack?

- The stack begins at the location pointer %esp points (top of stack), and all the addresses above it.

What is considered the current stack frame?

- The start of ebp and extending all the way to %esp. Stack frames are “chained”. What does this mean? How is it achieved?
- This means we can move on to previous stack frames within our program if we wanted to. We achieve this through the prologue: push %ebp, movl %esp, %ebp. How can you use this to say access the third argument of the function 4 calls prior to you.

```
movl %ebp, %eax # current stack pointer (address)
.rept 4
movl (%eax), %eax # dereference current address, and move on to the next pointer
.endr # 4 function calls are accessed before us
movl $4*wordsize(%eax), %ecx # access 3rd argument through advanced indexing mode
```

Write assembly instructions that emulate the effect of a push instruction.

```
subl $4, %esp # move pointer down 4 bytes (all pointers are 4 bytes)
movl src, (%esp) # moves src on the set up esp location (top of stack frame)
```

- What this means is that whenever we push some source into memory, we are decreasing the memory location of the stack pointer esp by 4 bytes of memory. Esp always points to the 'top of the stack'; but depending on what way you orient it, it can also be on the 'bottom of the stack'. However, it will always move towards the lower memory address (towards memory 0).

Write assembly instructions that emulate the effect of a pop instruction.

```
movl (%esp), dest # moving esp to dst (current stack)
addl $4, %esp # move pointer location up by 4 bytes
```

Write an assembly function that is callable from C that emulates the following C code.

```
short max(short* nums, int len){
    int index;
    short cur_max = nums[0];
    for( index = 1; index < len; index++)
    if( nums[i] > cur_max)
    cur_max = nums[i];
    return cur_max;
}
-> see asm references for solution
```

Write an assembly function that is callable from C that emulates the following C code.

```
short rec_max(short* nums, int len){
    short rest_max;
    if(len == 1)
        return nums[0];
    else{
        rest_max = rec_max(nums + 1, len -1);
        return nums[0] > rest_max ? nums[0] : rest_max;
    }
}
-> see asm references for solution
```

Final Review

Be able to convert hex to binary and vice versa.

- Binary to Hex:
  - Group the bits into 4 digits, starting from the right
  - Convert each group to decimal
  - For each decimal group, represent it in its hex form
    - 0 - 9: decimal, and 10 - 15 a - f respectively

Be able to convert a number from one base to any other base.

- Convert base 1 to decimal
- Continuously divide by base 2 until the quotient becomes zero.
- The solution will be the remainder written top-down, from right to left.

Know how signed integers can be represented, 2's compliment and sign magnitude.

- Unsigned:
  - Normal binary form
- Interpreting bits with 2's compliment:
  - If (MSB == 1); To represent the 2's complement number, negate the bits and add 1. Then represent read as a (-) binary value.
  - When we add in binary, one way of doing it can be to simply first convert binary to decimal and add 1 that way.
  - If (MSB == 0), then its signed, unsigned, and 2's complement expression will all be the same.
- Interpreting bits with Signed numbers:
  - If MSB == 1, then (-) value, and the MSB does not contribute to its magnitude.
  - If MSB == 0, then (+) value, and its read in its normal binary form.

Be able to convert a real number to its floating point representation and vice versa.

```
<31 sign><30----exp-----23><22------------mantissa-------------0>
```

1. Convert floating point to binary, by doing so independently for the part before or after the decimal.
2. Convert the binary expression into its proper form that is:
    i. 1.something
    ii. Multiply by 2^exp amount (shifting decimal place)
3. Exp += 127
4. The part after your decimal spot will be your mantissa.
5. Sign = 0, then +, otherwise -.
6. Convert exponent to binary
7. Fill in your mantissa will 0's, if need be to so that it fulls 22 bits total (starting from 0).
    i. The amount of zeros necessary to fill in == numBits(mantissa) - 23

Understand the two different ways of storing multidimensional arrays in memory: one contiguous chunk and arrays of arrays

- For one contiguous chunk
    - Know the difference between row major and column major
        - Both describe different ways of organizing multidimensionary arrays into linear memory storage.
        - Row Major: The consecutive elements of the rows are linear. Used in C.
        - Column Major: The consecutive elements of the columns are linear. Used in fortran and R.
    - Given an array and indices be able to generate the equivalent single dimensional address
        - Row Major: array[d1][d2][d3] == *(*(i * d2 * d3 + j)* d3 + k)
        - Column Major: array[d1][d2][d3] == *(*(k * d1 * d2 + j)* d1 + i)
    - Given an array and indices be able to generate assembly code that could access the array at those indices

// Defined on the stack

```
# short cur_max = nums[0];
movl nums(%ebp), %ecx #ecx = nums
movl (%ecx), %edx #edx = nums[0]
movl %edx, cur_max(%ebp) #cur_max = nums[0]
```

- For array of arrays:
    - Given an array and indices be able to generate assembly code that could access

the array at those indices

// Defined globally:

```
# eax is i
# ebx is j

movl a(, %eax, wordsize), %ecx #ecx = a[i]
movl (%ecx, %ebx, wordsize), %ecx #ecx = a[i][j]
```

// Defined on the stack

```
#ecx = i, #edx = j
movl a(%ebp), %ebx #ebx = a
movl (%ebx, %ecx, wordsize), %ebx #ebx = a[i]
movl (%ebx, %edx, wordsize), %ebx #ebx is a[i][j]
```

If a machine is little endian, what does that mean? If it is big endian?

- Endianess is the order of bytes represented by multidata/primative types such as int for float. The way/direction data is read can vary for different operating systems.
- Little Endian: The LS Byte is read first, and is stored in the lowest address. E.g. 0x4325296, 96 will be set to address 0.
- Big Endian: MS Byte is read first, and is stored in the lowest address. E.g. 0x4325296, 43 will be set to address 0.

Is Intel little endian or big endian?

- Little Endian

Given C code be able to translate it to assembly.

What are the gcc C calling conventions?

- Before calling a function, the arguments are pushed onto the stack, from right to left.
- EAX, ECX, EDX are not considered to have live values when a function is called.
  - Meaning it would be necessary to save these values before we call a function. This give the function the freedom to manipulate and use these registers, and once its finishes - it must restore these registers back to their original value.
- All registers besides EAX, ECX, and EDX are also expected to be saved before a C FUNCTION calls another function.

- Returned values are returned through eax.

What is considered the stack?

- Whatever esp points and the addresses above it.

All accesses to the stack must be of what size?

- Wordsize: The stack is an array of pointers, each of which have size 32 bits or 4 bytes for x86 machines. Therefore every byte within the computer has an id number, also known as an address.

What is considered the stack frame?

- All addresses of memory between EBP and ESP inclusive
- The current stack frame:
  - This partially depends on what defnition one chooses to use, but the book defnes it as starting at %ebp and extending to %esp.

(VERY IMPORTANT) Stack frames are chained, write assembly code to access the stack frame levels above the current stack frame

- What this means:
  - This means we can move on to previous stack frames within our program if we wanted to. We achieve this through the prologue: push %ebp, movl %esp, %ebp.
- You can, for example, use this to say access the third argument of the function 4 calls prior to you.

```
movl %ebp, %eax # current stack pointer (address)
.rept 4
movl (%eax), %eax # dereference current address, and move on to the next pointer
.endr # 4 function calls are accessed before us
movl $4*wordsize(%eax), %ecx # access 3rd argument here
```

Or two levels above the stack frame:

```
movl %ebp, %eax
movl (%eax), %eax
movl (%eax), %eax
```

Explain the difference between I/O mapped and Memory Mapped I/O

- I/O Mapped:
  - Separates the I/O space and memory space.
  - Different kinds of instructions are used to access I/O devices.
    - I/O: IN and OUT for read and write
    - Memory: MOV
  - Seperate bus lines: Used to distinguish memory reads and writes vs I/O reads and writes.
- Memory Mapped I/O:
  - All memory is within a unified space, but there is a portion in memory that is uniquely reserved for I/O.
  - Unlikely I/O Mapped, we use the same instructions to access both I/O and memory.
- Intel uses I/O mapped approach

Explain how I/O Polling works

- Is when the CPU repeated keeps asking the I/O device if it has input (*are you there, are you there...*), and once is eventually does it gets read by the CPU. The advantage of this is that it's really simply to implement. The disadvantage, that its really wasteful of CPU cycles.
  - What does the software do?
    - Basically Everything.
  - What does the hardware do?
    - Basically Nothing.

Explain how Interrupt driven I/O works

- Much smarter implementation of polling in that it is more efficient. However, more difficult in its implementation. I/O interrupts are handled on call, rather than just waiting on a call all the time.
  - What does the software do?
    - Sets up interrup vector table, and provides the code for the interrupt service routines.
    - It goes through process of fetch/decode/execute/store/check for interrupt. The 'check for interrupt is triggered by the I/O device, which then sets the interrupt line (between the CPU and I/O device) to true. Then the CPU will activate 'interupt acknowledged' in order to now service the I/O device. The device itself

replies with its identification number (IRQ). Then the (ISR) service handler will goes to this address, which is calculated by:

```
address = Interupt Distrupter Table + IRQ * 8
```

        - The interrupt diserupter contains the list of I/O devices, and there identifications.
    - What does the hardware do?
        - Provides capability to the software. Hardware interupts by the user can happen at any time, during any executing instruction.

What is Direct Accessed Memory? How is it helpful?

- DMA allows I/O devices to directly access memory.
- Helpful because it frees up the CPU to do other work while I/O is being preformed.
- Makes things more efficient by taking control over the bus lines. So, rather than read and writing to output and memory through the CPU, it does this through the DMA, letting the CPU do its own thing.

Given that the OS is just an ordinary program, what makes it “special?”

- What makes its 'special' is that it is the first program that is loaded into memory.

Explain how the OS loads a program into memory?

- /a.out written in shell program
- Location a.out in the file system by the OS.
- Look into the header file to determine how much memory is necessary to allocate to the program.
    - Check through the memory allocation table, which tells us what is in use and what is free to give.
    - Then load, and reserve the .text, .data, stack, and heap into memory, and mark it as 'in use'.
- Locate the beginning of a program, the address of which is located through the linker.
- Save all the registers (.. and later restore after program execution)
- Jump to the start of the program and begin execution.

Explain how the OS is loaded into memory.

- Turn on PC

- Program counter (PC) begins at 0xffffff0 in Intel.
- Runs the bootstrap loader @PC, which is statically stored in ROM. Loads section of the OS -> then hands off control to the OS.

What does it mean for a program to be running?

- That the PC is currently pointing into an instruction that belongs to your program.

Can the OS stop your program while it is running?

- No. Only one thing is running at once. If your program is running that your operating system isn't. However, your code can become halted if you voluntarily give up control by making a call whose instruction is located within the OS, or a hardware interrupt occurs - causing the CPU to jump to the IRS (located inside OS).

What does it mean if a program is marked as being in Sleep State?

- Context to the question:
  - Run and sleep states of programs are maintained by the process table, which maintains information about processes currently being run by the system, and whether or not the processes are runnable.
- Sleep means that the program cannot be currently run. In general, this happens when it is waiting for I/O to be preformed on its behalf. E.g. cin >>;

What does it mean if a program is marked as being in Run State?

- That the program is identified to be ELIGIBLE to run.

Explain what time sharing is and how it works.

- Programs will run one thing at a time at most. There an allocated amount of time that is reserved for every program. The process is switched very quickly between program making the illusion that things are being run simultaneously, or multitasking.
- E.g. Wifi can only service one person at a time. It switches from user to user, sending packets of information.
  - What does the software do?
    - A timer is also used to regulate interrupts periodically. Interrupts are what actually switch between programs processes, which requires saving the

registers of the current program, restoring the registers for the program to be run, and then jumping to the first instruction of the program to be run. Sets up and maintains process table. Provides the code to do the actual switching of the processes.

- What does the hardware do?
  - Provides the interrupt capabilities in CPU, to become able to switch to another task.

What are the principles that make caching work?

- The goal of cache index locations of memory in a smart way, because accessing memory can be 100 times slower than executing an instruction.
- Temporal and Spatial locality.
  - Temporal: Deals with how frequently variables are accessed. e.g. for loops
  - Spatial: Deals with the physical proximity between things. e.g. continuous chunk of array, local variables, classes, structs. And they tend to be accessed together, or at least next to each other.
    - Is determined by the line_size. Increasing the line size will also increase spatial locality.
- The principle of locality states that programs will spend 10% of their time on 90% of the code. This is do to a lot with repetitions from loops, and how frequently memory is accessed.
- Without cache, the CPU would have to search through a large memory base which could take a while. Cashes provides the solution of not being to large to look for, but small enough to do most of the work for you.

Where does the cache lie?

- Between the CPU and memory (RAM).

What does it mean for the cache to be transparent?

- That neither the CPU or memory know that cache is intermediately between them. CPU can function without cache.

Explain how the cache is structured.

- Always keeps a copy of a subsection of memory.
- Definitions:
  - Line: Continuous chunk of memory. This is what the cache reads/writes into.
  - C = size of cache in bytes

    - Line size (LS): number of bytes in a line.
    - Sets (S): number of groups of lines (not always contiguous).
    - #lines (NL): number of lines in cache.
    - Ways (W): number of lines in a set.
- Mapping Equations:
    - #lines (NL) = C/line_size
    - ways = #lines/set
    - sets = #lines/ways
- Where an address is located:
    - A set should contain element A that is floor(A/LS) mod S
    - Within the line itself, element A will be contained in position A mod LS.
- Address Partitioning:
    - Each line address is composed as follows:
    - <---tag bit---><---setBits---><---offsetBits---->
        - offset: How far into the line to look for your value
            - num bits = log_2(line_size)
        - set: identification on which set
            - num bits = log_2(sets)
        - tag: identification between lines in a set
            - left over bits or
            - address - #set bits – number of line offset bits
- Unique bit locations:
    - Set number = (A≫log2(LS)) & log2(S)
    - Offset = A & log2(LS)

What is the purpose of the Valid bit?

- Informs the CPU whether or not the bit is valid. That is to say where or not its a junk or not. When your computer turns on, it is invalid or 0, because it contains nothing. It is also set to zero when a context switch happens (switching between programs), because the data stored in it is not valid for the program that came in.
- It will be set to 1, when a read or write is performed on the line.

What is the purpose of the Tag bits?

- There are used to identify the line in cache, and how it maps to memory. Since cache in much smaller that memory, it is not a 1:1 correspondence. Instead, it identifies which line of memory is in cache. How this mapping works can vary. Strategies are discussed below.

What is the purpose of the Dirty bit?

- Are used with writeback policies. Write back policies are used when writing to memory from cache.
- We write to memory if and only if a collision occurred within the cache, meaning the location in cache has become overwritten, or changed. Collisions occur in cashe when cashe itself is full.
- On start up, dirty bit = 0
- 0 indicates that cashe and memory are the same. 1 indicates otherwise.
- If the dirty bit is 1 then the line in cashe has been written to, meaning we have to write to memory. We write to memory because we wrote something new to cashe, and therefore it must also be new to memory. Otherwise, its alway been there to begin with, so we wouldn't have to write to memory (0) = dirty.

What is the formula for determining what set a line is placed?

- floor(Address / line size) % Number of Sets

What is the formula for determining the line offset of an address?

- Address & line size

How many ways are there in a Direct Mapped cache?

- Number of ways = 1
- #sets = #ways = #lines

How many sets are there in a Fully Associative cache? In a Set Associative cache?

- In a fully associative cache, #sets = 1

Why are the cache's designed to have the number of Sets and line size as powers of 2?

- Because division would just meaning looking up the bits within the address, making things a lot quicker. So calculating things like address%lineSize, or (address/lineSize)%numberOfSets would simply mean look a the positions of the bits.
    - For context, division >> (right shifting) and multiplication << (left shifting)

(IMPORTANT) Be able to write inline assembly code. Explain what each of the following inline constraint modifiers do

- = : Used in output section. After the assembly code has become completed the value in this register will be copied to the associated C variable its been linked with.
-
    - : The linked variable is both an input and output. E.g. +b(bar); means bar = ebx
- & : Early clobber variable. The value will be written to before all the inputs are used up, forcing gcc to place it within its own register. Makes it so we use less registers, since it cannot be placed in same register as an input.
- a : Used to assign to a specific register -> eax (I prefer just using r)
- r : Let gcc chose the register for you

Translate the following C code into inline assembly

```
int* left_right_add(int* array, int len){
  int* result = malloc(len *sizeof(int));
  int index;
  result[0] = array[0] + array[1]
  result[len -1] = array[len-1] + array[len-2];
  for(index = 1; index < len; index++)
  result[i] = array[i-1] + array[i] + array[i + 1];
  return result;
  }
```

Midterm 1 Extras:

- Can represent 2^B objects with B number of bits.
    - equivocally, you can represent N number of objects with ceiling log_2(N) number of bits.
- A binary string cannot mean anything without the context of the problem. These are within the assembly with jg, jl, etc instructions.
- CPU Cycle:
    - Fetch, decode, execute, store.
- OS is used for:
    - memory control/allocation, file systems, system calls, and program communication with the OS.
    - I/O device management
    - Process management (starting programs, and time sharing)

    - Memory management: memory allocation table, whether or not memory is free.
- Shifting:

  Left Shift: (signed) 6 << 1 : add 1 zero from right

  ```
  - also use for multiplication
  - 6 << 3 = 6 * 2^3
  - does not wrap
  ```

  Logical right shift: (unsigned) 6 >> 1: add 1 zero from left

  ```
  - opposite to above
  - 12 >> 1 = 12 / 2^1
  - does not wrap
  ```

  Arithmetic right shift: (signed) -6 >> 2: add 2 ONES from left

  ```
  - sign preserving
  - -12 >> 2 = -12 / 2^2 = -3
  ```

- How does typing in a higher level langauge affect asm?
    - No type in asm, so nothing is going to stop from adding, for example an int and a char, something some higher level languages wouldn't allow. A higher level language will affect asm by the types you use. For example, using int or character would mean the allocation of 4 and 1 bytes respectfully. In addition, adding pointers together would also meaning pointer arithmetic, and using the addl instruction regardless on the type of data because all pointers are 32 bits or 8 bytes. Otherwise, we would use movb or addb, etc.
- Byte addressable:
    - Byte addressable means that the addresses for an instruction refer to individual bytes in memory.
- Name at least 2 reasons why a program compiled on one computer may not run on another computer.

    - Differing hardware. Remember that different CPUs might have completely different instruction sets — for example, what was an add instruction on an Intel CPU might mean something completely different on a MIPs CPU.
    - Different OS. Nearly all (if not all) programs rely on making system calls to the OS, but these system calls will differ across different operating systems. For example, the open() call for opening a file on a Unix system is not the same as the call for opening a file on a Windows system. If the operating systems involved are different, the program won’t work on both machines even if the hardware is identical.

Midterm 2 Extras:

- Understand how the location of a variable's declaration affects where it will be stored. I.e if a variable is local, global, or static where will space be made for it.

```
* Code: or .text. This is where the body of instructions your code contains.
* Data: or .data. Where both global and static variables are contained.
* Stack: holds local variables, arguments from functions, and their return addresse
* Heap: dynamic memory (i.e. with new in C++ or malloc).
```

  - If a function is recursive, then it would not be possible to convert local variables to global ones, because a recursive function in it's nature, every function call requires its own unique instance of the variable.
  - How advanced indexing is calculated:
    - displacement(base, index, scale)
    - Used to access the location in memory. Leal is used to access the address at that location. This is like doing movl $array, %eax. Here, we are not accessing any memory, but just storing the location or address of array into eax.
    - `displacement + base + index * scale`
    - defaults parameters are 0,0,0,1 respectively
    - leal will use the same formula. However, it only ever access the first address of example, [100-103], and not the full range.

    Examples: eax = 100, ecx = 5 (a) Movl (%eax), %ebx accesses addresses 100-103 (b) movb (%ecx,%eax), %ebx accesses address 105 (c) movl %eax, %ebx doesn't access anything at all! (d) movw (%ecx, %eax, 4) accesses addresses 405-408 (e) movl $someArray, %eax no mem access (f) leal anything; no mem access

- Emulating assembly instructions:
  - call push %eip jmp foo
  - ret pop %eip
  - pop movl (%esp), REGISTER addl $4, %esp
  - push subl $4, %esp movl VALUE, (%esp)
-

Final Extras:

- What is the use of input section in inline asm?
  - Before the program is run, the registers within the input section will be assigned to

their linked variable.

- What is the use of clobber section in inline asm?
  - Contains the lists of registers that are neither in the input or output. Instead, these are registers that you manipulate in the actual code part of your assembly. The purpose of the clobber is for the gcc to save these registers, and restore them back - in order to make sure things go back to the way they were before. We always place "cc" because this is the flags register, which always needs to be saved before hand.
- How fast is cache?
  - miss rate * memory access time + cache access time
- Advantages and disadvantages of cache:
  - advantages:
    - IT WORKS YO.
  - disadvantages:
    - If you have a lot of writes, and replace the same location in cache.
    - If you search through all of cache without finding anything.
- What is Direct and Fully Associative cache, and what are the Advantages / Disadvantages between them?
  - Direct Mapped:
    - This gives us a 1:1 direct correspondence between line in memory and line in cache. So the replacement policy is also like this.
    - Advantages:
      - Less expensive, smaller tags
      - Super fast: to locate O(1), but more collisions
    - Disadvantages:
      - 2 lines in memory that are accessed from the same line in cashe would require a swape in and out of cashe.
  - Fully Associative:
    - Any line in memory can be mapped to any line in cache.
    - Advantages:
      - Solve disadvantage of direct map.
    - Disadvantages:
      - O(N) search required, but less collisions.
      - More tag bits required, more expensive.
  - What is K-way set Associative cashe?
    - The same line of memory is always mapped to the same set in the cache but can be placed in any line contained within the set.
    - There are K ways per set. K is chosen by the designer of the cache
    - Defines set associative, and direct map with unique parameters.
- What are the replacement policies for set associative?

    - Least Recently Used: Remove the line in the set that was the least recently accessed
        - Makes the most sense but is too expensive to implement exactly for sets that have morethan 2 ways
    - Most Recently Used: Remove the line in the set that has been most recently accessed
        - This would be good for streaming
    - First in First out: Remove the line in the set that has been there the longest
        - Easy to implement
    - Random: randomly remove one
        - Simulations show that it only performs slightly worse than techniques based on usage
    - What is the memory tree hierarchy?
    - Sorted by priority: CPU (register manipulation), cache (locality), memory (RAM), disk (static memory). On the top of the hierarchy, things are more expensive, power hungry, and data sparse (less physical space to store byte)
- Advantages and disadvantages of write through or write back?
    - Write through:
        - Writing to cache always means writing to memory as well.
            - Advantages:
                - works, memory stays up to date.
                - a small number writes means
            - Disadvantages:
                - slow
    - Write back:
        - Advantages:
            - Every write to cache is not always written to in memory.
        - Disadvantages:
            - Extra memory dirty bit per address
- How does reading from address A cache work?
    - Locate the set.
    - Search through set by looking at the tag bits in every line.
    - Locate the line offset to locate the byte we want to return to the CPU.
    - If the search fails:
        - Remove a line from cache based. This is based off your replacement policy.
        - If dirty: write out to memory
        - Copy line containing address A into cache
        - Clear dirty bit and set the valid bit
        - Use the line offset to determine which byte in the line we want and return it to the CPU

- Writing to a Cache
  - Given that we want to write to address A
  - First use mapping equation to locate the set that could contain A
  - Search through this set comparing the tag bits of each line with the tag bits of the address
  - If there is a match with the tag bits then this line contains the value we are searching for
    - Use the line offset to determine which byte in the line we want to write to and do the write
  - If there is no match then that means the line is not in the cache
    - First choose a line in the set to remove based on the replacement policy
    - If that line is dirty write it out to memory
    - Copy the line that contains address A into the cache
    - Do the write to the cache
  - Set the valid bit
  - If using write back set the dirty bit to 1
  - If using write through write the value to memory as well
- Types of Misses:
  - Compulsory Miss:
    - First time accessing a line will be a miss
    - Increase line size to reduce this
  - Capacity miss:
    - Program execution overloads the cache size, so it requires to go out to memory, copy the stuff out, and then bring it back into cache repetitively.
  - Conflict miss:
    - When lines in memory map to the same set.
    - Increase number of ways per set to reduce this.