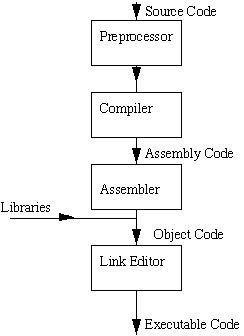
Computer Architecture and Assembly Language Class Notes

1/20/16

* A computer is any system that is comprises of hardware and the software that is running off of the hardware, aka the computer system
* A bit is a single universal piece of information for a computer (1 or 0)
* We group 8 bits together and call it a byte, a byte is a unit of information that is composed of 8 bits
* A word consists of either 4 bytes or 8 bytes, for 64 bit computers, words contain 8 bytes while 32 bit computers contain 4 bytes
* Stages of compilation:
  + Pre-processing
  + Compiler
  + Assembler
  + Linker
* Pre-processing: modifies the original source by replacing the shorthand names of functions with their definitions
* When you have a “#define PI 3.14” it replaces all instances of PI as 3.14?
* Pre-processing then turns the original source code into modified source code
* The human readable code is preprocesses and then is passed off to the compiler which generates an assembly file
* The compiler then turns out a hello.s file if it was given a hello.i file from the pre-processing procedure
* The next step is the file hello.s is given to the assembler where it turns the previous file into binary
* The assembler then sends the file to the linker where it gains it’s connections to the libraries and it is turned into an executable file.



* The operating system is the software that runs between the hardware and the basic software that can only be run on an OS
* The virtual address space is where the OS calls memory addresses,
* Program code is kept in the bottom of the VAS
* Shared libraries are in the middle of the VAS
* The heap is the space between program code and the shared libraries and it grows up
* The stack the space between the kernel and the shared library and grows downwards towards the shared library.
* Applications cannot access the kernel it is only there to maintain order for the OS
* When you call a function is it stored in the stack

1/25/16:

* C is called a procedural language
* “int main (void){ “ this is always necessary, it is always the starting point of a C program
* “int main (int argc , char \* argv[]){“ takes in a argument and returns an int
* #include brings in libraries while #define lets us input our own functions and lets us declare constants
* gcc –E hello.c will halt the compilation after the pre processing stage, you will get everything replaced by stdio.h or any constants that you had replaced.
* We can also make macro functions by #define SUM(A,B) ((A)+(B)), and in the program if you said SUM(PI,RADIUS) it would add ((3.14)+(4)).
* int c; on a 32-bit system will take up 4 bytes of space from the memory
* int c; are called standard identifiers
* we can freely cast between data types in C
* long data types take up 8 bytes of data
* short takes up 16 bytes of memory
* all operands are going to be implied as int
* binary operators are addition and subtraction
* unary operators will come first, they include negation(adding negatives) and other functions
* Unary operators in the same expression are executed from right to left
* Binary operators are evaluated from left to right
* x\*y\*z+a/b-c\*d, first the c\*d would be done then the a/b then the x\*y\*z, and then from left to right they would be summed
* z-(a+b/2)+w\*-y, first the b/2 will be evaluated then the a will be added, then the negation of y will be computed then the w\*-y, then all the binary functions will be evaluated form left to right
  + Example: #define SQUARE(s) ((s)\*(s))

1/SQUARE(3+4)

1/((3+4)\*(3+4))

* Printf(format\_string, values)
  + %d = int
  + %lf = double
  + %s = string
  + %c = character
  + %llu = long long
* The ampersand in the scanf points to the variable that you are planning on scanning into the program
* C also has logical operators such as &&, ||, ! which come out a binary response of either 0 or 1
* Arithmetic is evaluated first in an if statement then logical operator then relationships then equality and finally logical operators
  + Example:

Double x = 3.0;

Double y = 4.0;

Double z = 2.0;

Int flag = 0;

!flag || (y+z>=x-z)

* There are three types of if statements, if(), if() else(), and if() else if() else()
* You can only use switch statements with a character or integer
  + Int a = 6;

Switch(a){

Case 0:

Process\_zero();

Break;

Case 1:

Process\_one();

Break;

Default:

Process\_no();

}

1/27/16:

* when you are entering in integers in #define, make sure that if a decimal is possible you put 2.0 instead of 2 so that it can be considered a double.
* If you are writing a basic C function, there is no need for extra parentheses unlike macros
* If you have an array of size 2 you are guaranteed that element 0 is next to element 1
* How to declare an array: double dvals[16]
* Arrays in C are just pointers to their memory locations
* If you printf(“%lf”, dvals[7]) you get whatever was left in that memory location beforehand.
* If you printf(“%lf”,dvals[16]) you get whatever is in the memory space one space above where the array actually ends and you will probably get a piece of random data.
  + Int my\_int\_array[6];
  + Scanf(“%d”,&my\_int\_array[3]);
* My\_func(int); changes made inside a function are local, they do not affect anything outside of the function unless it is an array where it will be affected globally. (passed by reference)
* Chat ma[2][5]; the array will create a block of 10 memory locations but it knows where the new row is set
* We can create complex data types with “structs”
* Typedef: he C programming language provides a keyword called typedef, which you can use to give a type, a new name. Following is an example to define a term BYTE for one-byte numbers
* Typedef also lets you build structs
* Typedef is processed in the compiler, not the pre-processor like most other basic data types
* The compiler deals with a line of a program when there is a semi colon at the end of it
  + Typedef struct {
  + Char student[50];
  + Int total\_hw\_points;
  + Int assignments\_collected;
  + } student\_hw\_t;
  + student\_hw\_t mharmon;
  + mharmon.total\_hw\_points = 0;
  + mharmon.assignments\_collected = 0;
  + mharmon.student[0] = ‘\0’;
  + strcpy(mharmon.student,”mharmon”);
  + void int sum (int,int\*);
  + int sum(int a, int \*b){ // the star represents that it is a pointer to another integer
  + \*b = a + \*b;
  + }
* to call sum in a main function we do: sum(a, &b);
* The & takes a variable and asks for the memory address of the variable that is attached on to
* The \* reads the information that is stored in a memory address \*
  + Int a = 7;
  + Int \*b = &a;

2/3/16

int \*ip; /\* pointer to an integer \*/

double \*dp; /\* pointer to a double \*/

float \*fp; /\* pointer to a float \*/

char \*ch /\* pointer to a character \*/

* For the first lab start with tinycalc.c and tinycalc.h and then work through each function one by one
* Start with check\_command, the only purpose of this is to see if tinycalc can actually do something
* ..finally access the value at the address available in the pointer variable. This is done by using unary operator **\*** that returns the value of the variable located at the address specified by its operand.

#include <stdio.h>

int main () {

int var = 20; /\* actual variable declaration \*/

int \*ip; /\* pointer variable declaration \*/

ip = &var; /\* store address of var in pointer variable\*/

printf("Address of var variable: %x\n", &var );

/\* address stored in pointer variable \*/

printf("Address stored in ip variable: %x\n", ip );

/\* access the value using the pointer \*/

printf("Value of \*ip variable: %d\n", \*ip );

return 0;

}

* When the above code is compiled it output’s:

Address of var variable: bffd8b3c

Address stored in ip variable: bffd8b3c

Value of \*ip variable: 20

* It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.
* We use pointers to make changes to variables outside of the functions, we can return information by a function call without using a return statement and instead we can pass the pointers as variables.
* To assign a pointer to a variable, use an ampersand with the variable's name.
* Though pointers are declared with an asterisk they are not always used with an asterisk.

|  |  |  |
| --- | --- | --- |
| **Pointer Thing** | **Memory Address** | **Memory Contents** |
| p | Yep | Nope |
| \*p | Nope | Yep |
| \*p++ | Incremented after value is read | Unchanged |
| \*(p++) | Incremented after value is read | Unchanged |
| (\*p)++ | Unchanged | Incremented after it's used |
| \*++p | Incremented before value is read | Unchanged |
| \*(++p) | Incremented before value is read | Unchanged |
| ++\*p | Unchanged | Incremented before it's used |
| ++(\*p) | Unchanged | Incremented before it's used |
| p\*++ | Not a pointer | Not a pointer |
| p++\* | Not a pointer | Not a pointer |
| **Array Notation** | **Pointer Equivalent** |
| array[0] | \*a |
| array[1] | \*(a+1) |
| array[2] | \*(a+2) |
| array[3] | \*(a+3) |
| array[x] | \*(a+x) |
| **Declaring** | | |
| type \*x; | Pointers have a data type like normal variables. | |
| void \*v; | They can also have an incomplete type. Operators other than assignment cannot be applied as the length of the type is unknown. | |
| struct type \*y; | A data structure pointer. | |
| type z[]; | An array/­string name can be used as a pointer to the first array element. | |
| **Accessing** | | |
| x | A memory address. | |
| \*x | Value stored at that address. | |
| y->a | Value stored in structure pointer y member a. | |
| &varName | Memory address of normal variable varName. | |
| \*(type \*)v | Dereferencing a void pointer as a type pointer. | |

int array[100];

int \*ip = array; // both of these now point to the same location in memory

for(i=0;i<100;i++){

printf(“%d\n”,\*(ip+1));

printf(“%d\n”, array[i]);//these above statements are the same

}

Feb 10

1. Word size
   1. number of bytes that = “a word”
   2. determines size of data types
   3. ie: void \*p; (32 bit system)
      1. size = 4 bytes
   4. ie: char \*p; (34 bit system)
      1. size = 4 bytes
   5. pointers have to be big enough to point to anything in memory
   6. 16 bit system
      1. 2^bitwidth - 1
   7. size will be independent of what it points to. it holds a memory location!
   8. ie: int a - 32 bit -
   9. ie: long - 64 bit -
2. Endianness
   1. (not really a problem working in C or Java)
   2. **byte** storage ordering (which end of the egg to crack first)
      1. systems do it differently
      2. some order most important or least important
      3. some order least important to most important
   3. big endian -most significant first
   4. little endian -least significant first
   5. ie: ox12345678
      1. big endian -> 12345678
      2. little endian -> 78563412 (bytes are backwards not bits)
   6. app programmers don't really need to worry about it
   7. only when you need to disassemble your code (needed for sending code through networks)
   8. standard is big endian
   9. show\_bytes.c
      1. give it some memory address
      2. prints out the hex value for that address
      3. you can see what endian your system uses
3. boolean algebra and operation
   1. truth tables
      1. useful way to evaluate logic
   2. AND
      1. truth table = p^q

|  |  |  |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

* 1. NOT

|  |  |  |
| --- | --- | --- |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

* 1. OR

|  |  |  |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

* 1. XOR - exclusive or

|  |  |  |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

* 1. combination of boolean operations is what the computer uses to perform algebra
  2. p^(p+q)

|  |  |  |  |
| --- | --- | --- | --- |
| p | q | p+q | p^(p+q) |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |

* 1. Identities ( ~ = not, /\ = and, \/ = or)
     1. Identity law
        1. AND: p\/1 = P
        2. OR: p\/0=p
     2. Dominance law
        1. AND: p/\0 = 0
        2. OR: p\/1=1
     3. Idempotent
        1. AND: p/\p = p
        2. OR: p\/p = p
     4. Compliment
        1. AND: p/\~p=0
        2. OR: p\/!p=1
     5. 2xCompliment
        1. AND ~(~p)=p
     6. Commuting
        1. AND: p/\e = e/\p
        2. OR: p\/e = e\/p
     7. Associative
        1. AND: p/\(q/\r) = (p/\q)/\r
        2. OR: p\/(q\/r) = (p\/q)\/r
     8. Distributive
        1. AND: p/\(q\/r) = p/\q + p/\r
        2. OR: p\/(q/\r) = p\/q + p\/r
     9. DeMorgans
        1. AND: ~(p/\q) = ~p\/~q
        2. OR: ~(p\/q) = ~p/\~q
     10. Absorption
         1. AND: p/\(p\/q) = p
         2. OR: p\/(p/\q) = p
  2. Conditional expressions
  3. bit level operations
     1. & AND | OR ~ NOT /\ XOR
  4. && logical operator
     1. int a = 99: ->1
     2. int b = 0; ->0
     3. a&&b -> 0
     4. applies to the whole expression
  5. ie: oxA4 & oxC6
     1. looks bit by bit!
     2. binary:
     3. 10100100
     4. 11000110
     5. 10000100 = ox somethin

1. Shift operations
   1. take a value and shift the binary some number left or right
   2. generally truncated (with some complications)
   3. left shift
      1. easier
      2. shift all bits to the left
      3. drop the most significant bit(s)
      4. trial with zeros
      5. ie: ox FF = 1111 1111
         1. char a = oxFF;
         2. a<<3;
         3. a=1111 1100
   4. right shift
      1. SPACES ARE FILLED WITH MOST SIGNIFICANT
      2. ie: ox 0F = 0000 1111
2. Shift operations
   1. take a value and shift the binary some number left or right
   2. generally truncated (with some complications)
   3. left shift
      1. easier
      2. shift all bits to the left
      3. drop the most significant bit(s)
      4. trial with zeros
      5. ie: ox FF = 1111 1111
         1. char a = oxFF;
         2. a<<3;
         3. a=1111 1100
   4. right shift
      1. SPACES ARE FILLED WITH MOST SIGNIFICANT
      2. ie: ox 0F = 0000 1111
         1. char a = ox 0F
         2. a>>3;
         3. a = 0001 1110
      3. logical
         1. most significant are filled by zeros
      4. arithmetic
         1. most significant are filled by whatever the most significant value was
         2. represents the sign of the value so we can maintain it
      5. NEVER KNOW WHICH
         1. arithmetic is more common
         2. never assume!

Feb 15

linked list - type \* list

* 1. shift past the end?
     1. mod by word length
     2. ie: 32 bit shift by 35 -> shifts by 3 (35%32)
  2. ie: 1 << 2 + 3 << 4
     1. order of operations?
     2. addition is first!
     3. (1<<2) + ( 3<<4)
  3. ie: 0101 << 1 = 1010
  4. 5 -> 10
     1. shift left by 1 multiply by 2
     2. shift right by 1 divides by 2
  5. bit vector
     1. **V** = {v(w-1) , v(w-2), … , v1, v0}
     2. largest amount of unsigned integers = 2^w-1
     3. ie: B2U (bit to unsigned) = sum from i=0 to (w-1) of vi \* 2^i
     4. sign is in most significant bit
        1. most negative = 1000 0000
        2. least negative = 1111 1111
        3. only seven bits to represent positive
     5. now max unsigned = 2^(w-1) -1
        1. min = -2^(w-1)
        2. \*\*\* zero is positive \*\*\*
     6. “Two’s Complement”
        1. 1111 1110 = -2
        2. 1111 1111 = -1
     7. B2T (binary to twos compliment) = -v(w-1) \* 2^(w-1) \* sum from i=0 to w-2 of vi \*2i
     8. conversions
        1. signed to unsigned
        2. nothing changes? bit pattern the same

2/17/16:

int in\_range(int v){

if(v>=1 && v<=100)

return 1;

return 0;

}

* we can optimize this code with only one operator by using a compiler trick:

if((unsigned int)(v-1)<=99)

* If you put in 0 it turns v-1 into -1 and it turns -1 into an unsigned int which is a very very large positive number
* All negative numbers are the signed world are largest in the unsigned world
* Subtraction with binary
* For addition subtraction and multiplication the integer sizes are limited by their bit system
* Overflow is the concept that happens when you try obtaining a number that is bigger than the allotted memory
* For unsigned operations C doesn’t have a guarantee for what it will give out, but the assumption is that it will most likely give out the mod of that number
* For signed operations C will take the mod of the number
* Computers define different steps for signed and unsigned
* Unsigned:
* For x + y where the addition of the tow is less than the word size it is just computed normally
* When x + y is greater than or equal to the word size but less than the 2^(w+1) you have x + y – 2^w
* Signed:
* When you have x+y that is less than the word size you take no extra steps
* When you have x+y that is greater than or equal to the word size but less that the 2^(w-1) you take x+y-(2^w)
* But when you have negative overflow you have take x+y+2^w when x+y is less than -2^(w-1)
* when you have a signed integer that has negative overflow it will always be interpreted as a negative because there will always be a one as the most significant bit

2/22/16:

* to use the twos compliment to negate binary words, you flip all 0’s and 1’s and then you add a one to the least significant bit
* When you multiply binary words, if you have a 1\*1 you always carry a one to the next column
* In multiplication, if you have a overflow you mod the number by 2 to the word length

X Y x\*y x\*y(truncated)

101 011 1111 111

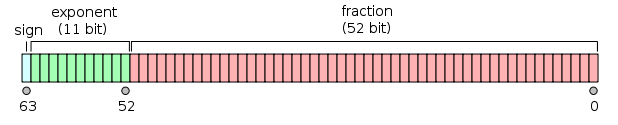
5 3 15 7

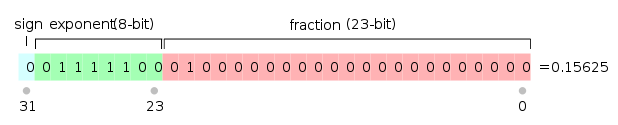
-3 3 -9 111

110111 -1

* For a single precision float you only have 32 bits while in double precision you have 64 bits to store the number
* 31.4159 is the same as 3.14159\*10^1
* D =
* (b represents the binary integer in that spot)
* one bit is represented as the sign of the number so there are 31 bits that are available to be used
* there are bits that are used for the exponent and there are bits that are used to represent the siginificant number
* the mantissa is the part of the floating point number that is after the decimal point
* you never need more than one sign bit for signle precision or double precision
* we use eleven bits for the exponent for double precision and 8 bits for the exponent of a 32 bit number
* all of this goes from most significant bit to least siginificant
* we have 52 bits for the mantissa and 23 bits for the mantissa on a double precision number

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a [technical standard](https://en.wikipedia.org/wiki/Technical_standard) for [floating-point](https://en.wikipedia.org/wiki/Floating_point) computation established in 1985 by the [Institute of Electrical and Electronics Engineers](https://en.wikipedia.org/wiki/Institute_of_Electrical_and_Electronics_Engineers) (IEEE)





* to get the actual exponent value for the the number you translate the binary into decimal and then you minus the exponent by 2^(e-1)-1
* 0 is astually a signed number, you can havea positive or a negative 0

2/24/16:

* There ar three methods of estimation for floating point numbers
  + Round to even: 1.40 becomes 1, 1.60 becomes 2, 1.50 becomes 2 2.50 becomes 3 and -1.50 becomes -2. This way of rounding will always to an evne number if it can
  + Round to zero: All the numbers will round towards 0
  + Round up: All the numbers will round up towards positive infinity so -1.50 would round to -1
  + Round down: You will round down towards negative infinity
* Round even is the default way for floating point numbers to be rounded
* To compile, first it gets sent to the pre-processor, then it gives us hello.i which puts in the direct code from #include, then it goes to the compiler and gives us hello.s and then we have our assembler which turns it into machine code and the linker that brings in another .o and gives us an executable at the end
* Gcc –O0 will turn all optimizing off and it wont make any changes to your code
* Gcc-O0 –m32 will compile the program in 32bit
* Gcc –O0 –m32 –S hello.c will output your .s file that is created by the compiler (it will stop after the compilation process)
* ISA (instruction set architecture) Every CPU has its own set of ISA
* In [computing](https://en.wikipedia.org/wiki/Computing), an opcode (abbreviated from operation code) is the portion of a [machine language](https://en.wikipedia.org/wiki/Machine_code) instruction that specifies the operation to be performed. Beside the opcode itself, [instructions](https://en.wikipedia.org/wiki/Instruction_%28computer_science%29) usually specify the data they will process, in form of [operands](https://en.wikipedia.org/wiki/Operand).
* When there are parenthesis in assembly xode ,it means that you are dealing with a memory location
* There are 4 classes of operation:
  + Load
  + Operational
  + Computational
  + Flow control
* Ia32 defines a set of 32bit registers:
  + %eax %ah %al
  + %ebx %bh %bl
  + %ecx %ch %cl
  + %edx %dh %dl
  + %esi %sh %sl
  + %edi
  + %ebp
  + %esp
* The first 4 are for accessing information as a full 32bit or a different type of bit configuration
* The remaining are to be used as full 32 or 16 bit
* The first six are general purpose which can mostly be used without restriction
* %eax is considered the accumulator, it is used for loops generally
* Some functins will use %eax as a destination for the result of a function call
* %ebp is the frame pointer, it keeps the memory address of the current stack
* if you have 8(%ebp),%eax it moves the stack frame up 8 bytes and that is where the %ebp will be now
* %esp is our stak pointer, it points to the bottom of the stack
* %ebp is the upper bound and %esp is the lower bound
* if you have:
  + subl %esp, $40 it moves the stack pointer 40 bytes down so you can store more information
* we use the $ sign to refer to an actual integer so if you have $7 it translates to 0x7, it designates immediate value
  + addl $4, %eax will add 4 to the value of the register %eax
* The third way to specifiy is with memory
  + The parenthesis indicate that we are doing something with a memory location
* %aex and %edx are the only registries used for multiplication and division
* When %eip is corrupted (when it points to a memory address that does not acutally contain any program code the CPU will not be able to fetch legitimate code to execute so the program will crash.

3/2/16:

* The $ represents immediate value, you would use this to add $8 to some address
* Parentheses represent memory locations
* And % represent registries
* These are the three types of data that can be represented in assembly
* On pg. 169 figure 3.3 shows all forms supported
* R = looks at the registry
* Example: Imm(Ex) $8(%eax) 🡪M[8+&[%eax]]
* (,Ex,s) (,%eax,2) 🡪M[2\*R[%eax]]
* (Ev,Ey,s) (%eax, %ebx, 4)🡪M[R[%eax] + 4\*R[%ebx]]

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Location | 0x100 | 0x104 | 0x108 | 0x10c | %eax | %ecx | %edx |
| Values | 0x000000FF | 0x000000AB | 0x00000013 | 0x00000011 | 0x100 | 0x1 | 0x3 |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

* + %eax = 0x100
  + 0x104 = 0xAB
  + $0x104 = 0x104
  + (%eax) = 0xFF
  + 4(%eax) = M[4+R[%eax]+4\*R[%edx]]
* The most common type of assembly code that we are going to see is the movement of data
* We have 3 types of ways to move data:
  + - **The move instruction: MOV**
    - The \_ will either be:
      * l = 4 bytes
      * w = 2 bytes
      * b = 1 byte
    - MOVL $99, %eax : moves the immediate value 99 into the registry %eax
    - MOVL (%edx),%ecx : moves the memory address of %edx to %ecx, so that %ecx contains the memory address of %edx
    - You cant do MOVL (%eax), (%edx)
    - to get around this you do:
      * MOVL (%eax), %ecx
      * MOVL %ecx, (%ebx)
    - MOVW %bp, %sp will move two bytes
* **To move only 2 bytes you take away the preceding “E” of the registry**
  + - MOVB (%ebp), %ah
    - MOVS moves something smaller into something bigger, it takes the most significant bit and extends it until it fits the size
    - MOVZ moves something smaller into something bigger, it extends the data with 0’s
    - %dh = 0xCD
    - %eax = 0x98765432
    - MOVB %dh, %al
    - %eax would now equal 0x987654CD
    - MOVB %dh, %al
    - MOVSBL %dh, %eax
    - MOVZBL %dh, %eax
    - now %eax is equal to 0x000000CD
* variables in the programs are kept from %ebp down to the top of the stack so the first variable would be stored in %ebp-4, then %ebp-8
* any parameters in the program are kept from %ebp up, the first parameter would be stored at %ebp+4, then %ebp+8
* The purpose of %esp is to point to the top-most item of the stack
* If the program has declared 3 variables %esp will point to %ebp-16

**AT&T SYNTAX(SWITCHED FOR INTEL SYNTAX)**

* movl source, destination // copies 32-bit value from source into destination // no effect on value of source
* leal (src1, src2), dst // dst = src2 + src1
* addl rightop, leftop // leftop = leftop + rightop
* subl rightop, leftop // leftop = leftop – rightop
* imull rightop, leftop // leftop = leftop \* rightop
* negl op // op = -op
* incl op // op = op + 1
* decl op // op = op – 1

**MIDTERM TOPICS:**

* **Malloc**
* **Pointers**
* **Under the hood of what happens in our code (recursion)**
* **Broad picture of our architecture**
* **How data is represented**
* **Some assembly language**
* **Data movement with data movement instructions**
* **Compilation process**
* **True false questions (8-10 questions)**
* **2-3 creative questions**
* **the remaining will be stuff like the homework**

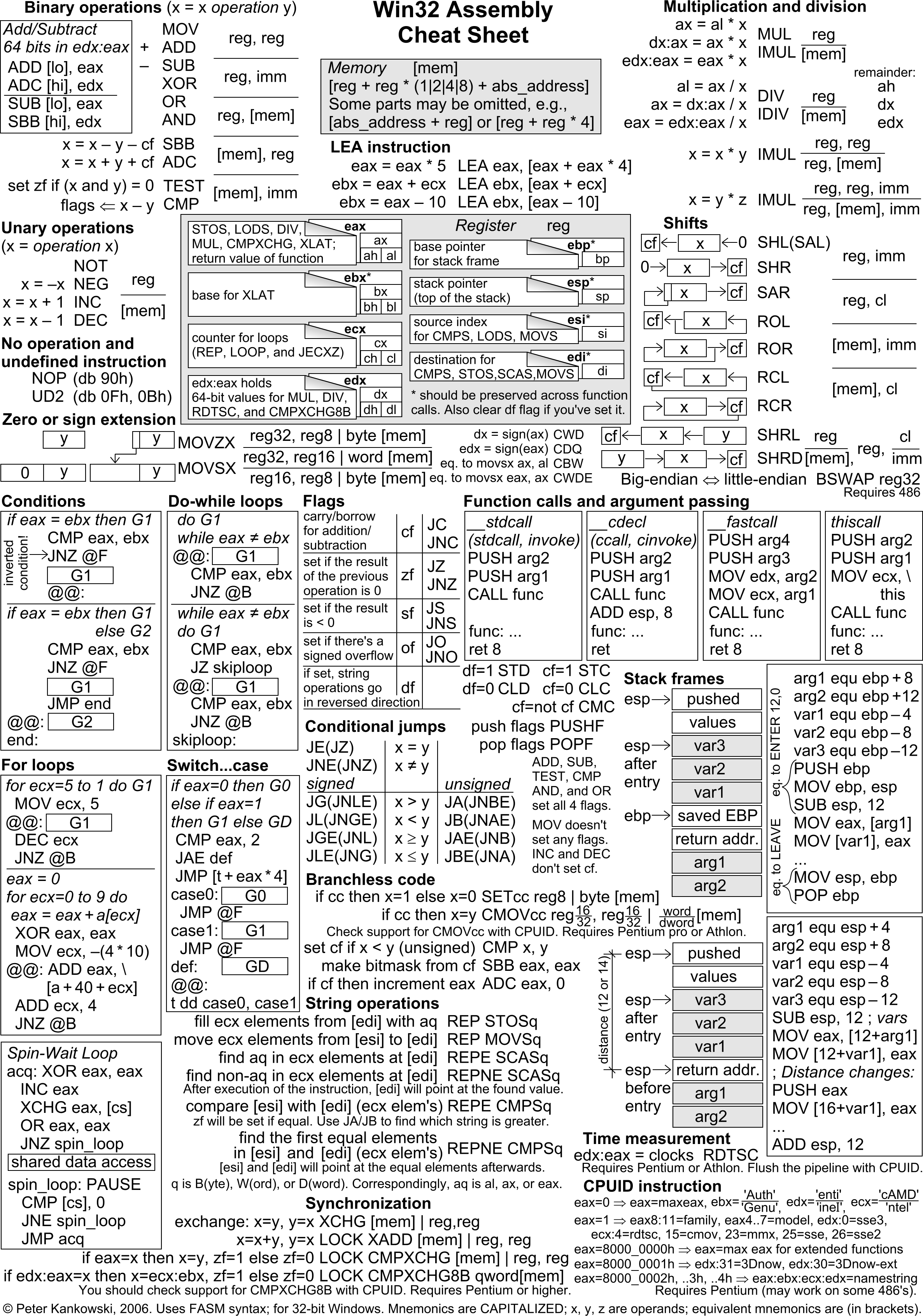
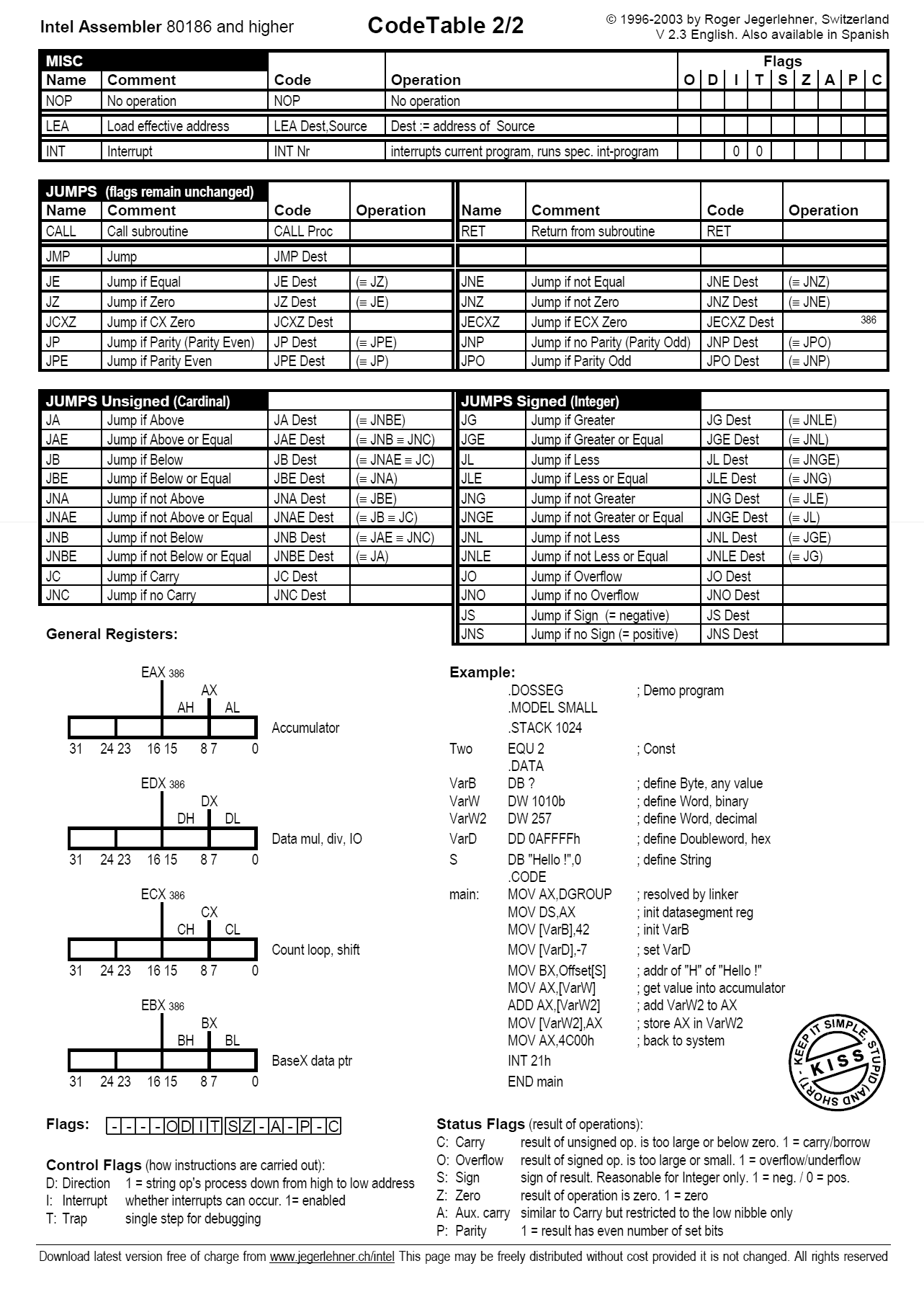
3/7/16:

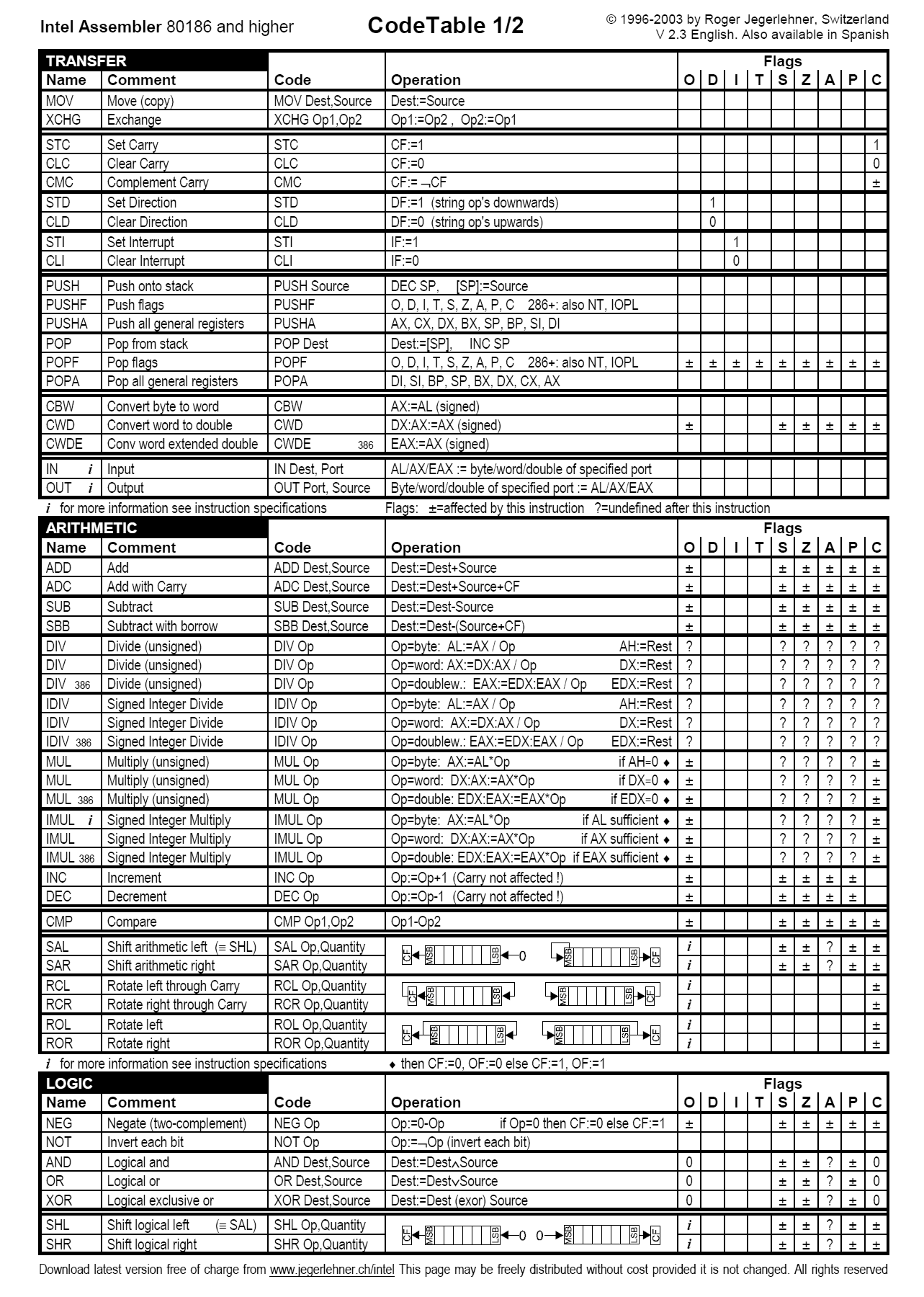
* pushl Rx will put a new value (Rx) onto the stack, it will decrement the stack pointer by 4
  + pushl Rx
  + subl $4, %esp
  + movl %eax, (%esp)
* popl Rxwill push the top value off of the stack and move the stack pointer up by 4
  + popl Rx
  + movl (%esp), %eax
  + addl $4, %esp

**STUDY GUIDE**

* for preprocessor functions use #define (int a, int b) ((b\*h)(2.0))
* when you have \*&c1, that is legal because you are getting the value of the memory location of c1
* cannot take the & of a pointer so:
  + char cp2, cp1;
  + cp2 = &cp1; THIS DOES NOT WORK
* NEVER FORGET THE NULL CHARACTER WHEN MALLOCING SPACE FOR STRINGS
* From H to D:
  + Take the farthest number to the right and multiply it like so:
    - Ex: 0xDF23
    - (3\*16^0)+(2\*16^1)+(16\*16^2)+(13\*16^3)
    - to get 57123
* From D to H
  + Divde my 16 and take the remainders and reverse the order
    - Ex: 23834
    - 23834/16 = 1484 R 10 (A)
    - 1484/16 = 93 R 1 (1)
    - 93/16 = 5 R 13 (D)
    - 5/16 = 0 R 5 (5)
    - now read bottom to top for the remainders
    - to get 0x5D1A
* logical shifting is padded with zeros and arithmetic trail with the most significant (the far left bit)
* left shift is always logical
* right shift can be arithmetic or logical
* for signed and unsigned ints:
  + Ex: 0x50
  + Bx0101 0000
  + 80 in unsigned and signed
* sometimes when you have signed ints:
  + Ex: 0xF2
  + Bx1111 0010
  + Unsigned is 242
  + For signed you ignore the first bit and only use that for negative or positive
  + So signed you convert 0111 0010 to 242
  + Then you minus 2 to the power of the word size in bits so if the word size is one you do 2^8 so for this example it would be 242-256 to get -14
* if asked for x + y, you can just add the integers
* if asked for unsigned x + y, convert the numbers to bits and if there is overflow just cut it off after the word size so:
  + Ex: 1100 + 0100 = 10000
  + So you cut off the first 1 so it would be 0 in unsigned

3/14/16:

* We have subtraction, negation, and leal, in the ALU we generally use leal to compute pointers
* 
* 



* Leal doesn’t occur in conditional statements, mostly used in conditionals are jmp’s, and cmp’s
* %eax = x

%ecx = y

leal (%eax,%ecx), %edx **This is equal to x + y**

movl (%eax,%ecx), %edx **This stores the data of whatever is in the memory location x + y**

%edx = 2y+x+9

leal 9(%eax,%ecx,2),%edx **This is equal to 2 \* %ecx plus %eax plus 9**

* %eax = 0x100

%ecx = 0x1

%edx = 0x3

0x100 = 0xFF

0x104 = 0xAB

0x108 = 0x13

0x10C = 0x11

addl %ecx,(%eax)

M[R[%eax]] = R[%ecx] + M[R[%eax}}

(0x1)+M[0x100]

0x100 = (0x1)+(0xFF)

* subl %edx,4(%eax)

4(%eax) = 4(%eax) - %edx

M[R[eax]] = [R[%eax+4]-R[edx]

M[(0x100)+4] – (0x3)

M[0x104]

(0xAB)-(0x3)

= 0xA8

**The memory address 0x108 would now contain the value 0xA8**

* Decl %ecx

R[%ecx] = R[%ecx]-1

* Imull $16,(%eax,%edx,4)

M[4\*r[%edx]+R[%eax]] = m\*16

4\*0x3 = 0xC

M[0x10C]\*16

(0x11)\*(0x10) = 0x110

**In the memory location 0x10C the value is now 0x110**

* Addl %eax, $12

**Cant have an immediate value as a destination, this does not work**

* SHL stands for left shift and SAL is als for left shifting
* SHR is for logical right shifting
* SAR is for arithmetic right shifting
* Logical shifting fills with zero’s
* While arithmetic fills with the least significant digit
* [INST] k, D
* K is the amount you are shifting and D is the address of what you are shifting

SHLL $2, %eax

SHLL $32, %eax **this will produce zero**

* Int Shift\_left2right(int x, int n){

X << = 2;

X >> = n;

Return x;

}

* **%ebp is the frame pointer**
* Codition Codes (Flags)
  + CF - Carry Flag – used for detecting overflow for unsigned integers
  + ZF- Zero Flag – The zero flag is set to one if the last ALU operation resulted in a 0
  + SF – Sign Flag - Sign flag is set to one if the most recent operation is a negative
  + OF – Overflow Flag – This is for signed overflow (2’s complement overflow)
* There is cmp, compare acutally does a subtraction and chekcs if the number is 0 and if so it will set the ZF to 1.

CMP S2, S1 **performs s1-s2**

TEST S2, S1 **performs a bitwise AND so s1 & s2**

SET D

SETE %al, **it will set %al to 1 if the last operation proved to be equal**

* C to Goto code:
* Goto mylabel: code
* Shifts code to where label is
* Ugly code but it works

3/28/16:

* Procedure call:
* We use the stack to:
  + Arguments
  + Save program counter state
  + Save local state
    - Pointers to places on the stack that you will need again
    - A procedure needs the stakc plus registers because there are only 8 registers and any space you malloced in the function
  + Save register state
* %ebp is the stack frame pointer
* Caller/callee convetions:
  + Caller save registers are %eax, %edx and %ecx
  + Callee can change the but it doesn’t change the stack
  + %esi, %edi and %ebx, these are the callee save registers

Int Caller(){

Int arg1 = 534;

Int arg2 = 1037;

Int sum = swap\_add(&arg1, &arg2);

Int diff = arg1 – arg2;

Return sum \* diff;

}

Int swap\_add(int \*xp, int \*yp){

Int x = \*xp;

Int y = \*yp;

\*xp = y;

\*yp = x;

return x + y;

}

2/30/16:

* Arrays are just pointers of that are set up in a way that so that each element is after the other in memory
* Every element if 4 spaces down the heap
* Arrays are contiguous memory
* Xa + i\*L to get the memory location of an array element
* Xa is the starting location of the array
* i is the index of the array
* L is the size in bytes of data type T

Elem Size Total Size Start Address Element at i

Short s[7] 2 14 S0 S0+2i

Short \*T[3] 4 12 T0 T + 4i

Long double v[8] 12 96 V0 V0 +12i

Long double \*w[8] 4 32 P0 P0+4i

* (Ra, Rb, S) = Rb \* s + Ra
* Ra is Xa
* Rb is i
* S is L

Int \*p;

P+i = p + 4i

= P + L\*i

Given E = array of some type stored in %edx = E0

i = %ecx

* Int a[3][5] is equal to:

Typedef int row\_t[3]{

Row\_t A[5];

}row\_t[3];

* TD[R][C];
* TR[i][j];
* X0 + L(C\*i+j);
* This is called row major ordering, if your columns are 3 you have move three places across the memory to get to the next row
* 4/6/16:

Loop 1:

for(i=0;i<sz;i++){

if(vals[i]&0x1=0x1)

sum++;

movl -8(%ebp), %eax

leal 0(,%eax,4), %edx

movl 8(%ebp), %eax

addl %edx, %eax

movl (%eax), %eax

andl $1, %eax

testl %eax, %eax

je .L3

addl $1

Loop 2:

For(i=0;i<sz;i++){

a=(vals[i]&0x1==0x1);

Sum+=a;

movl -8(%ebp), %eax

leal 0(,%eax,4), %edx

movl 8(%ebp), %eax

addl %edx, %eax

movl (%eax), %eax

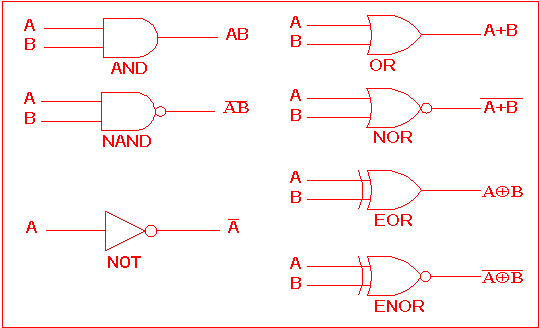
andl $1, %eax

movl %eax, -12(%ebp)

movl -12(%ebp),%eax

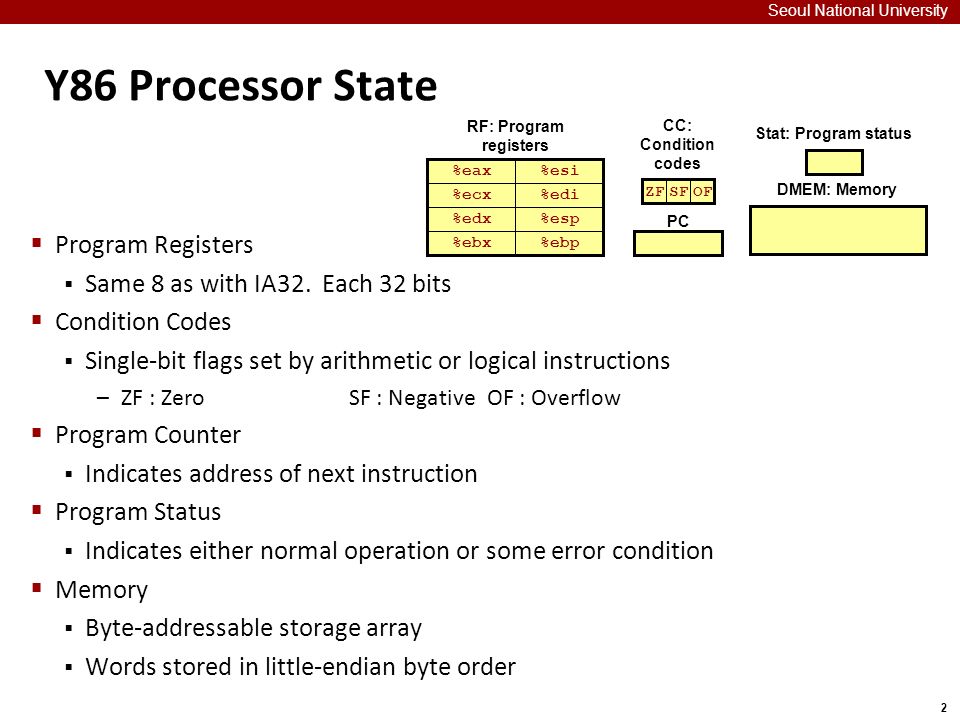
addl %eax, -4(%ebp)

* we have 3 mov instructions, 4 alu instructions and 1 jump instructions in loop 1
* we have 5 mov instructions, 4 alu instructions and 0 jump instructions in loop 2
* Jumps make code slow!
* In computer architecture, a branch predictor is a digital circuit that tries to guess which way a branch (e.g. an if-then-else structure) will go before this is known for sure. The purpose of the branch predictor is to improve the flow in the instruction pipeline.
* An instruction cycle (sometimes called fetch-decode-execute cycle) is the basic operation cycle of a computer. It is the process by which a computer retrieves a program instruction from its memory, determines what actions the instruction requires, and carries out those actions.
* The first phase of the instruction cycle is called fetch
* The cpu will fethcthe bytes that comprise the instruction from meory and load them into the instruction register
* The program counter holds the address of the next instruction
* Phase 2 of the instruction cycle is to decode
* It determines if they are memory address, immediate values or registries
* Phase 3 of the instruction cycle is to execute
* Run the instructions I nthe instruction register and using the operands given it executes the instruction
* After the execute phase there is the memory phase where we read and write data from memory
* Then it is followed by write back where registries are updated
* The last phase of the instruction cycle is to update the program counter
* RISC: RISC (reduced instruction set computer) is a microprocessor that is designed to perform a smaller number of types of computer instructions so that it can operate at a higher speed (perform more millions of instructions per second, or MIPS).
* CISC: Complex instruction set computing (CISC /ˈsɪsk/) is a processor design where single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions.
* In a RISC setting we will find fixed length encoding, we will see that the instruction encoding are a fixed length
* CISC will have variable length encoding
* In RISC we often have very simple format, we have a base and an offset, we don’t have any scaling factors
* In RISC ALU operations only operate on registers
* CISC will have a lot more instructions compared to RISC
* RISC will keep things very simple but might not be faster
* The process of reading and writing data looks like a wave, it saves information until it is full and then in writes the data so the data that is saves disappears



* 0 = a&&b = AND
* 0 = a||b = OR
* !a = NOT
* we create a combination ciricuit when we but these logic gates together
* The ouput of 2 or more logic gates cannot be connected
* Circuits must be acyclic
* Y86 is the toy version of x86
* Y86 has RISC elements to it
* In y86 we still hve our 8 general purpose registers but we don’t have low byte registers
* We only have 3 conditional flags, ZF, SF, OF
* We don’t have unsigned variables
* We will have our program counter
* STAT are status registers, they maintain the status of our program
* We treat memory as a contiguous array (same as usual)

4/11/16:



* + %eax 0
  + %ecx 1
  + %edx 2
  + %ebx 3
  + %esp 4
  + %ebp 5
  + %esi 6
  + %edi 7
  + no Reg. F
* variable length encoding for y86 range from 1 byte to 6 bytes
* y86 is also little endian encoded

Y86 Instruction names: Encoding

* + Halt 00
  + Nop 10
  + Rrmovl rA,rB 20 rA rB
  + Irmovl V, rA 30 F rB
  + Rmmvol D, rB 40 F rB (mem address is represented as little endian)
  + Mrmovl D(rB), rA 50 rA rB
  + Opx addl rA, rB 6[x] rA rB
  + Jmx D 7[x] D
  + Cmovlle rA, rB 2[x] rA rB
  + Call D 80 D
  + Ret 90
  + Push rA A0 rAF
  + Pop rA B0 rBF

April 13

Lab hints:    Off by one

        Test other things not just the given one

        Try edge cases

1. Decode
   1. Get register values
   2. Uses a multiplex to select the signals you need to perform an operation
2. ALU (Execute phase)
   1. Arithmetic and logic
   2. ValE
      1. The value propagated by the ALU
      2. Contains the result of operations
   3. ValA and ValC
      1. Go into the same input path
      2. Needs a multiplex to select which one to use
   4. ValB
      1. Has its own input path
   5. icode and ifun go into the ALU first and tell it which inputs to use
   6. Condition codes are also outputs of the ALU (they are aware of the inputs too)
   7. Computes effective memory addresses
      1. Get values from the memory locations that are passed
   8. increments/decrements stack pointer
   9. Evaluates jump operations
3. Ie:     subl    %eax, %ecx
   1. ValA: 6
   2. ValB: 9
   3. ALU:
      1. ValE <- ValB-ValA
      2. ValE <- 3
      3. Set condition flags: ZF <- 0, OF <- 0, SF <- 0
4. Ie:     rmmovl    rA, D(rB)
   1. From fetch:
      1. rA, rB, ValC, ValP, icode, ifun
   2. From Decode:
      1. ValA, ValB
   3. In ALU:
      1. ValE = ValC + ValB
5. Register to register does not use the ALU
6. Memory Operations
   1. ValE and ValA or ValP are inputs (memory addresses)
   2. Icode and ifun are used
   3. ValM outputs
7. Write back
   1. Register files are updated
   2. Takes in ValE and ValM
   3. Outputs into two destinations (destE and destM)
   4. Icode and ifun tell you which register to adjust
8. PC update
   1. Takes in ValP, ValC, icode, ifun, condition codes, ValM
   2. At the return statement (encoded as 90):
      1. Fetch:
         1. Set icode = 9
         2. Ifun = 0
         3. rA =  /
         4. rB = /
         5. ValC: /
         6. ValP: PC + 1
      2. Decode:
         1. ValA: R[%esp]
         2. ValB: R[%esp]
         3. Need both! One to do memory access(readback) and one to update(writeback)
      3. Execute:
         1. ValE <- ValB + 4
         2. Changes the value of %esp
      4. Memory
         1. ValM <- M[ValA]
      5. Write back
         1. Pop = decrement %esp
         2. %esp = ValE
      6. PC update
         1. PC <- ValM

New topic!

1. Pipeline - doing more with less time
   1. Some things can begin running on certain components while other stuff is working on other components

April 18

1. Lab code y86
   1. Written in a simulator
   2. Given template to mimic the virtual address space
   3. Begins at .pos 0
   4. Sets the initial stack and frame pointer
   5. Then calls main
   6. The stack:
      1. .align 4 pads the zeros in groups of 4 so there’s no weird indexing
      2. Begins at zero ends at 0xFF0
      3. Define constants
         1. Array length
         2. Data (iterate by 4!)
         3. The names can be alias’ for locations
   7. Main
      1. Stuff then end with ret
      2. “Call” pushes that onto stack then jumps there
      3. Add our own function calls after the return statement in main BEFORE the stack
   8. Sort
      1. There will always be a sorted part of the array and an unsorted part of the array
      2. Step 1: assume the item in the first part of the unsorted section is the minimum
      3. Step 2: compare that value with everything in the section, if smaller, then swap
      4. Step 3: the next value we assume to be the minimum and compare it with each of the remaining values, if smaller, swap it
      5. Etc…
      6. TIP: write in C first then translate it to goto C then to IA32 then y86
2. Pipeline - Time or latency of an instruction
   1. Latency
      1. Time to accomplish a task
   2. Throughput
      1. Number of tasks per unit time
   3. Latency stays the same but changing the throughput changes the efficiency
   4. Pipelining seeks to increase the throughput
   5. If latency increases (by design of the chip) need to re adjust the pipeline
   6. Ie: sequential
      1. 320 ps (ps = picoseconds 10^(-12))
      2. 3.12 instructions / ns
      3. 3.12 billion instructions (GIPS = giga-instructions per second)
   7. To pipeline, break up the instruction set and introduce PIPELINE REGISTERS to keep track the state of the machine
   8. Break after decode, break after memoryc break at end = 3 phase pipeline
   9. Each section has a certain latency
   10. Somehow changes the latency and now uses giga-ops (operations)
   11. Potential for speeding up is limited to the operations
   12. Overall improvement is limited by the slowest operation
   13. Two instructions going through at the same time “compete for resources” (ie: accessing memory)
       1. Need to stall! And wait for others to finish!
3. balanced/uniform pipeline
   1. Uncommon
   2. Ie: 3 stage pipeline
   3. total latency: 360ps
   4. ie:

A: 80ps - B: 30ps - C:60ps - D:50ps - E:70ps - F10ps (assume 20 ps for registers)

How do you create a 2 stage pipeline? Where do you put the registers?

Limiting factor? - only as small as the longest instruction (in this case 80ps)

Clock cycle = 190 ps

Latency = 2 \* 190 = 380

5.26 Gigaops

1. Overlapping
   1. Can't start something before you have the information propagated by the previous operation
   2. Pipeline registers will capture the data and you can use it!
   3. Can't update the program counter before the end….
   4. Branch prediction
      1. Assume the code will go down certain branches
      2. If you're wrong you have to flush the pipeline and start over
      3. That gives you a time penalty!
      4. Hazards - data dependency - must introduce stall
   5. Ie:

5 stage pipeline (PC and WB are together)

Irmovl        $50, %eax

Addl        %eax, %ebx

Mrmovl        100(%ebx), %edx

        F  D  E  M  WB

            F  D

Problem in the third instruction! (decode)

Don’t have valE to writeback

    To fix: delay the fetch by one clock cycle so we can forward valE

April 20

1. Pipeline cont…
   1. Control hazard
      1. Ie:

    Subl     %eax, %ecx

    Je    goequal

    irmovl     $1, %edx

* + 1. Draw a pipeline diagram where you assume a 5-stage pipeline w/ writeback and program counter together

            F    D    E\_\_\_\_\_M    WB/PC

                F    D    E    M    WB/PC

                    F    D    E    M    WB/PC

* + 1. \_\_\_ - Where things break! Because of the subtraction and the jump. We need to know if we jump before we execute. Introduce a stall!!
    2. Need two stalls before we fetch the last time
    3. Lets use branch prediction!
    4. Two options:
       1. Branch one: jump
       2. Branch two: not jump
    5. “Always take” policy
       1. Always take the jump
       2. Worry about being wrong later
       3. 60% correct!
    6. Ie: when jump is followed

Subl     %eax, %ecx

    Je    goequal

    irmovl     $1, %edx

Irmovl    $0, %esi

…

Goequal.

Addl    %esi, %edx

Push    %edp

* + 1. Pipeline diagram

    Subl    F    D    E    M    WB/PC

    Je        F    D    E    M    WB/PC

    Addl            F    D    E    M    WB/PC

    Pushl                F    D    E    M    WB/PC

* + 1. \_\_- where we know for sure if we actually need to jump
    2. If it’s wrong: you need to stall and wait for the addl and pushl to leave the pipeline before you correct it (flush it out)
  1. Ways to minimize these hazards
     1. Delayed branching
        1. Used when you have code that must be completed regardless what happens to the branch
        2. This delays the branch execution
        3. Reduces time penalties
        4. Compiler optimization
     2. Refactor code to avoid conditional jumps

1. How to measure pipeline performance
   1. Cycles per instruction (CPI)
   2. Theoretical limit CPI = 1.0
   3. Any CPI = 1.0 +(∑ [ instruction rate \* freq\_penality \* penalty])
   4. Ie:
      1. Load errors are 25% of your errors with 20% of them data hazards - 1 clock cycle stall
      2. Branch errors are 20% with 40% failure rate - 2 clock cycles stall
      3. Return errors are 2% - 3 clock cycles stall
      4. CPI = 1.0 + (.25\*.20\*1) + (.20\*.40\*2) + (.02\*1.0\*3) = 1.27
      5. New branch prediction!
      6. Branch errors are 20% with 35% failure rate - 2 CC
      7. New CPI = 1.14

NEW/LAST TOPIC

1. Memory!
   1. Hierarchy

* 1. Cache
     1. Smaller and faster memory that we will use soon
     2. Organization
     3. Block size
        1. M = # SOMETHING
        2. 32 bit
     4. Block sets
        1. B = 2^b
     5. Cache sets
        1. S = 2^s
     6. Cache lines
        1. E
     7. Cache size
        1. C = B\*S\*E
     8. Use caching
        1. Use the addressing bit size to encode a given information with a valid code to indicate spot in memory
        2. Each block looks like:
        3. Each block is a set of binary values and the information code tells you how to access it
        4. The tag tells you if the information is in the right place / is this the information I am looking for??
     9. Ie:

    What is the number of sets? Block offset? Tag bits?

    Know:     size (C) = 1024 bytes

        M = 32

        Block size (B) = 4

        Cache lines per set (E) = 4 lines

    S = 1024 / 16 = 64

    64 = 2^s -> s = 6

    4 = 2^b -> b = 2

    The rest must be tag bits! 24

* + 1. In order to access memory, you need a valid bit set to 1 and the same tag bits before you can read the values in memory
    2. Types of caching:
       1. Direct mapped cache
          1. Only one line per set (E = 1)
          2. Selected set is the selected cache line
          3. Simplest implementation
       2. Set-associative cache
          1. Each set has more than one cache line
          2. 1 < E < C/B
          3. Ie: 2-way set associative cache (E = 2)
          4. Benefit: fixes address conflicts with memory with similar bit patterns
       3. Fully-associative cache
          1. Only one set with tons of cache lines
          2. Used for very small things (L1)
          3. E = C/B
          4. Can be slow if used for anything bigger because the computer needs to check each line
    3. Eviction from cache
       1. Different ways (3 main approaches)
       2. Get rid of bit that was  used least recent
       3. Least frequency
       4. random!

4/25/16:

* while pipelining might make the throughput of a system quicker, it can also increase the latency of a single user
* In computer science, locality of reference, also known as the principle of locality, is a term for the phenomenon in which the same values, or related storage locations, are frequently accessed. There are two basic types of reference locality – temporal and spatial locality. Temporal locality refers to the reuse of specific data, and/or resources, within a relatively small time duration. Spatial locality refers to the use of data elements within relatively close storage locations. Sequential locality, a special case of spatial locality, occurs when data elements are arranged and accessed linearly, such as, traversing the elements in a one-dimensional array.
* the stride of an array (also referred to as increment, pitch or step size) is the number of locations in memory between beginnings of successive [array](https://en.wikipedia.org/wiki/Array_data_structure) elements, measured in [bytes](https://en.wikipedia.org/wiki/Byte) or in units of the size of the array's elements. The stride cannot be smaller than the element size but can be larger, indicating extra space between elements

Typedef struct{

Int val[3];

Int acc[3];

}point;

point p[N];

clear(int \*p, int n){

for(i=0;i<n;i++){

for(j=0;j<3;j++){

p[i].val[j] = 0;

for(k=0;k<3;k++){

p[i].acc[k] = 0;

}

}

}

i0🡪i1🡪i2🡪i3🡪i4🡪i5

int a[M][N];

for(j=0;j<N;j++){

for(i=0;i<M;i++){

sum+=[i][j];

}

}

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | J = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| I=0 | Miss | . | . | . | Miss |  |  |  |
| I=1 | Miss | . | . | . | Miss |  |  |  |
| I=2 | Miss | . | . | . | Miss |  |  |  |
| I=3 | Miss | . | . | . | Miss |  |  |  |

* caches move memory around in blocks of data
* Read Through -  reading a word from main memory to CPU
* No Read Through - reading a block from main memory to cache and then from cache to CPU
* Write Through - the information is written to both the block in the cache and to the block in the lower-level memory.   
  Advantage:   
   - read miss never results in writes to main memory   
   - easy to implement   
   - main memory always has the most current copy of the data (consistent)   
  Disadvantage:   
   - write is slower   
   - every write needs a main memory access   
   - as a result uses more memory bandwidth
* Write back - the information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced. To reduce the frequency of writing back blocks on replacement, a dirty bit is commonly used. This status bit indicates whether the block is dirty (modified while in the cache) or clean (not modified). If it is clean the block is not written on a miss.   
  Advantage:   
   - writes occur at the speed of the cache memory   
   - multiple writes within a block require only one write to main memory   
   - as a result uses less memory bandwidth   
  Disadvantage:   
    - harder to implement   
    - main memory is not always consistent with cache   
    - reads that result in replacement may cause writes of dirty blocks to main memory
* Write Allocate - the block is loaded on a write miss, followed by the write-hit action.   
  No Write Allocate - the block is modified in the main memory and not loaded into the cache.
* Temporal locality is the notion that if you reference code you will most likely reference it again later
* When registers don’t have avaue they are assumed as 0

5/27/16:

Virtual memory

* 1. Why this kind of complexity? Why not physical memory addresses?
  2. Advantages: the operating system manages the physical addresses so you don't overwrite key programs (called sandboxing : keeping processes separate)
  3. Sharing libraries is helpful (linking nicely!)
  4. Use main memory as somewhat of a cache for areas of memory that are active
  5. Page faults/pages
     1. CPU needs a section called Memory Management Unit (MMU) where virtual address space is loaded into
     2. The MMU translates this into physical addresses
     3. then loads into cache/memory spots if necessary
     4. MMU has N-bit addressable memory space
        1. Ie: 2^32 = 4GB
     5. Partitions memory into pages of equal size
        1. P = 2^p bytes
     6. Each maps directly one to one to a physical page (page frame in main memory)
     7. 3 different separate states
        1. Unallocated state
           1. Not in use/no data associated with it
           2. Not occupying space on disk
        2. Cached state
           1. Page is allocated AND in physical memory
        3. Uncached state
           1. Have been allocated but NOT in physical memory
           2. Only exist on disk
     8. Ie: Pages look like:

Virtual N=2^n              Physical M=2^m

|  |  |  |
| --- | --- | --- |
| VP0 |  | PP0 |
| VP1 |  | PP1 |
| VP2 |  | PP2 |
| …. |  | ... |
| VP2^(N-P)-2 |  |  |
| vp2^(N-P)-1 |  | PP2^(M-P)-1 |

* + 1. Page tables
       1. Keep track of mapping
       2. PTE (page table entry)
       3. Keep reference for every page
    2. PTE
       1. Valid bit
          1. Either unallocated or not cached (0)
          2. Allocated (1)
       2. Address
          1. If valid bit (0) and address is null = unallocated
          2. If valid bit (0) and address is a thing = uncached
          3. If valid bit (1) cached in main memory and the address referenced the physical memory address
       3. Ie: split VA into 8 PTEs what are the page states?

Virtual memory            Physical memory    Disk

PTE0:        0    NULL        PP0:    VP1        0xX\_\_1:  VP1

PTE1:        1    PP\_0        PP1:    VP2        0xX\_\_2:  VP2

PTE2:        1    PP\_1        PP2:    VP7        0xX\_\_3:  VP3

PTE3:        0    0xX\_\_3    PP3:    VP4        0xX\_\_4:  VP7

PTE4:        1    PP\_3                    0xX\_\_5:  VP6

PTE5:        0    NULL                    0xX\_\_6:  VP4

PTE6:        0    0xX\_\_5

PTE7:        1    PP\_2

* + 1. Page faults
       1. Trying to access memory that hasn't been loaded into the main memory but is mapped
       2. The operating system chooses to evict something to load the info we need
       3. Ie: if we tried to access PTE3, we would evict one of the PP’s and pulling in the info from the disk
  1. Canary value
     1. Buffer overflow attack protection
     2. known values that are placed between a buffer and control data on the stack to monitor buffer overflows. When the buffer overflows, the first data to be corrupted will usually be the canary, and a failed verification of the canary data is therefore an alert of an overflow

1. Dynamic memory allocation and Malloc
   1. Brk = break
      1. Special value
      2. In Y86 = .pos BRK
      3. Used to end your code
      4. Keep you from writing into important program code
   2. Use an allocator to monitor where memory allocation is allowed and the break is
      1. Explicit allocator
         1. Used in C
         2. Programmer has to explicitly free memory
      2. Implicit allocator
         1. Java/C#
         2. Garbage collector that de-allocates things for you
   3. Memory mapping (alternative to malloc)
      1. Function that returns a pointer to where your memory is allocated
      2. Void \* mmap(NULL, Given a size, read or write, file on disk?, file descriptor, offset)
      3. You can allocate space of any size
      4. Literally splits off a separate segment (unlike malloc)
      5. munmap(pointer) frees that space and the information actually no longer exists
         1. Segfaults if tried to access again (unlike malloc/free)
   4. Program break
      1. Used by malloc
      2. Void & sbrk(int)
      3. dynamically change the amount of space allocated for the data segment of the calling process. The change is made by resetting the program break of the process, which determines the maximum space that can be allocated
      4. Used if you run out of space, you can move the program break
   5. All about balancing throughput with memory utilization
   6. Fragmentation
      1. Internal
         1. The amount of memory we asked for was not on the double word boundary
         2. We end up losing some
      2. External
         1. If we free memory the next malloc call does not recognize the newly empty space
         2. Instead just grows the break
      3. Four ways to avoid fragmentation
         1. Data structure used to monitor free
         2. Placement
            1. There might be more than one free space!
            2. How do you choose??
         3. Splitting
            1. Split the free block into different sections to use
         4. Coalescing
            1. What do you do with free blocks?
            2. Combine consecutive free blocks into larger free sections