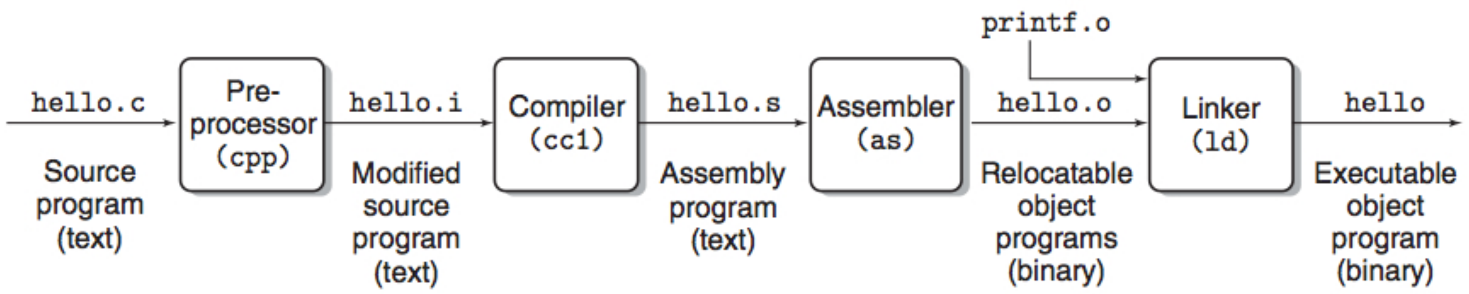
Assembly Languages

Recall the compilation process:



We can halt the compilation process at the assembly language step using some gcc flags:

```
gcc -O1 -S asm.c
```

This will generate asm.s as output and this file will contain assembly code. Computer systems execute **machine code** which are sequences of bytes that encode the operations supported by the system's architecture. **assembly language** code is a human-readable representation of machine code listing the individual intructions of machine code.

Example:

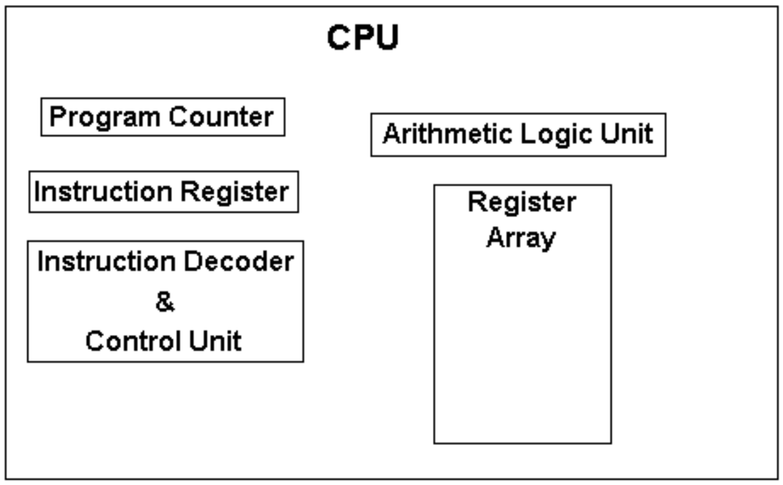
Machine code is just a stream of bits:

```
 F0 B8 00 00 00 09 B9 00 00 00 07 A8 89 98 E0
```

The machine code represents a sequence of operations. Its not good for reading eh? Assembly code is human-readble. Here is a generic assembly code for the machine code:

```
  PUSH          pc
  LD            a, 9
  LD            b, 7
  ADD           a,b
  RET           a
  POP           pc
```

Instruction Set Architecture



Each assembly language instruction in translated into machine code by the compiler. The format and behavior of the machine code is defined by the **instruction set architecture** (ISA) of the computer system. The ISA is unique to each chip architecture. This is why, unlike in Java, C code compiled on our lab machines will not run on your laptop. The ISA of our lab systems are likely quite different than the ISA on your laptop. In addition to providing operations, an ISA also makes internal states of the process accessible to the programmer. Typical this includes:

- the **register files** which contain named locations on the processor for sotring information. (we call each individual location a register). Because integer operations are quite different from floating point operations, a system that supports both will have a set of integer registers and a set of floating point registers. Some registers are reserved for holding program state and other execution critical information.
- the **program counter** maintains the address in memory of the next instruction to be exectued.
- the *conditional register* maintains status about the most recent arithmetic or logical operation. If, for example, an arithmetic overflow occured the conditional register might be set to indicate so.
- the *frame or stack pointer* is a special register that points to the location in memory that represents the top of the program stack.

  Instruction sets formats very widely from implementation to implementation. We can investigate the machine code with the assembly code above and make some assumptions about the ISA:

```
Machine                 ASM

  F0                    PUSH    pc
  B8 00 00 00 09        LD      a, 9
  B9 00 00 00 07        LD      b, 7
  A8 89                 ADD     a,b
  98                    RET     a
  E0                    POP     pc
```

  What can we glean about the ISA?

  8 bits for op code, 8 bits for registers, at least a 32-bit system. And so on.

Elements of an Instruction

The particulars vary from system to system (as always!!), but in general each instruction in an ISA will specify:

- **Operation Code**:(op code) tells the computer what to do.
- **Source Operand(s)**: tells the computer the data on which to perform the operation.
- **Result Operand**: tells the computer where to store the result. (This often implicit to the operation and the result is stored in one of the source operands)
- **Cycles**: the number of *clock cycles* required to complete the operation.

Example: A simple instruction format:

8 bits	**12 bits**	**12 bits**
Op Code	Operand Reference	Operand Reference
Instruction		

How many possible op codes? 2^8 = 256 unique operations

How much memory can we directly access? 2^12 = 4kB

Example: C and ASM comparison.

Consider the following C code:

```
int sum_it(int *xp, int y) {
  int t = *xp + y;
  *xp = t;
  return t;
 }
```

the equivalent Intel I32 architecture ASM might look something like this:

```
sum_it:
  pushl         %ebp            // save frame location from frame register
  movl          %esp, %ebp      // save new frame location to frame register
  movl          8(%ebp), %edx   // load *xp into edx register
  movl          12(%ebp), %eax  // load y into %eax register
  addl          (%edx), %eax    // add value at (%edx) to %eax, save to %eax register
  movl          %eax, (%edx)    // save value in %eax to location in %edx register
  popl          %ebp            // restore original frame location to frame register
  ret                           // return from sub-routine
```

Simple Recursion Example with Nano ISA

We'll use an imaginary ISA called Nano ISA to demonstrate what happens inside the computer system when we perform a recursive function call. We'll define "Nano ISA" as follows:

- **jp**: is the jump pointer register and holds an address of program code.
- **pc**: is the program counter register and holds the address of the next instruction.
- **rr**: is the return register, used to hold return values.
- **r1** and **r2**: are integer registers for holding integer vlues.
- Direct memory addressing is not supported
- Maintains a conditional flag register that updates each time a conditional operation is performed

With the following instructions:

- PUSH {OPERAND} - pushes the operand's value to the stack
- POP {OPERAND} - removes the top value on the stack and stores it in the register described by the operand
- DONE - indicates program completion, the value in *rr* is returned
- J {OPERAND} - jumps execution to the program location supplied by the operand. The value of *pc* is pushed to the stack.
- CMP {OPERAND} {OPERAND} - compares the values of the two operands and sets the conditional flag as (equal, less than or greater than).
- B{LT|GT|GTE|LTE|NEQ|EQ} {OPERAND}- conditional branch jumps executiuon to the location designated by the operand based on the conditional flag.
- ADD,SUB,MULT,DIV {OPERAND} {OPERAND} - performs the indicated arithmetic on the two operands and stores the result in the first operand.a
- RET - jumps execution to the value stored in *jp*
- MOV {OPERAND} {OPERAND} - moves the value of the 2nd operand into the first operand.

Lets look at a C function for computing the % of two positive integers

```
int main(void) {
  return mod(7, 3);
}

int mod(int x, int y) {
  if (x < y) return x;
  return mod(x-y, y);
}
```

Our Nano ISA assembly code might look something like this:

```
.main
1001            PUSH    (7)
1002            PUSH    (3)
1003            J       .mod
1004            DONE

.mod
1005            POP     jp
1006            POP     r2
1007            POP     r1
1008            CMP     r1,r2
1009            BLT     .L0
1010            SUB     r1,r2
1011            PUSH    jp
```

```
1012            PUSH    r1
1013            PUSH    r2
1014            J       .mod
1015            POP     jp
1016            RET

.L0
1017            MOV     rr,r1
1018            RET
```

we can trace execution line-by-line:

Line	Stack	pc	jp	r1	r2	rr	cf
1001	7	1002	?	?	?	?	-
1002	7,3	1003	?	?	?	?	-
1003	7,3,1004	1005	?	?	?	?	-
1005	7,3	1006	1004	?	?	?	-
1006	7	1007	1004	?	3	?	-
1007	-----	1008	1004	7	3	?	-
1008	-----	1009	1004	7	3	?	GT
1009	-----	1010	1004	7	3	?	GT
1010	-----	1011	1004	4	3	?	GT
1011	1004	1012	1004	4	3	?	GT
1012	1004,4	1013	1004	4	3	?	GT
1013	1004,4,3	1014	1004	4	3	?	GT
1014	1004,4,3, 1015	1005	1004	4	3	?	GT
1005	1004,4,3	1006	1015	4	3	?	GT
1006	1004,4	1007	1015	4	3	?	GT
1007	1004	1008	1015	4	3	?	GT
1008	1004	1009	1015	4	3	?	GT
1009	1004	1010	1015	4	3	?	GT
1010	1004	1011	1015	1	3	?	GT
1011	1004,1015	1012	1015	1	3	?	GT
1012	1004,1015, 1	1013	1015	1	3	?	GT
1013	1004,1015 1,3	1014	1015	1	3	?	GT
1014	1004,1015, 1,3, 1015	1005	1015	1	3	?	GT
1005	1004,1015, 1,3	1006	1015	1	3	?	GT
1006	1004,1015, 1	1007	1015	1	3	?	GT
1007	1004,1015	1008	1015	1	3	?	GT
1008	1004,1015	1009	1015	1	3	?	LT
1009	1004,1015	1017	1015	1	3	?	LT
1017	1004,1015	1018	1015	1	3	1	LT
1018	1004,1015	1015	1015	1	3	1	LT
1015	1004	1016	1015	1	3	1	LT
1016	1004	1015	1015	1	3	1	LT
1015	-----	1016	1004	1	3	1	LT
1016	-----	1004	1004	1	3	1	LT
1004	----	----	1004	1	3	1	LT

IA 23

IA-32 (Intel Architecture 32-bit) is one of the most common ISA in modern computer systems. We (and our book) will use it as the default assembly language in this course.

IA-32 defines 8 32-bit registers. they are:

- **%eax, %ebx, %ecx, %edx, %esi, %edi**: general purpose registers. All six may, for the most part, be used without restriction. This comes with the caveat that each does have specified purpose, so some intrsutions in the ISA may assume an operation for a given register. For example, %eax is considered the *accumulator* for storing operands and results data. Some arithmetic instructions, may implicitly store the results of operations in %eax.
- **%ebp**: is the *frame register* or *frame pointer*. It holds the memory address to the current stack frame. In other words the first memory location in the current stack context.
- **%esp**: is the *stack register* or *stack pointer*. It holds the memory address to the top of the stack.In other words the next

available memory location on the top of the stack.

Specifying Values

Remember each instruction in an ISA has an operator and a set of operands. There are three ways to specify operands in IA-32:

1. **immediate** values: immediate values are constants and designated with $ followd by an integer in C a valid C number format.
   Example:$1 // the integer value 1 (0x1)
2. **register** values: for accessing data stored in registers.
   Example: %eax // the value stored in the register %eax
3. **memory** values: for accessing memory locations. Memory values may be accessed *directly*, *indirectly* such as through a register (ex: (%eax),or through *effective addressing* where we perform some type of operation to compute the memory address.
   Example:4(%esp) // the memory address of the top of the stack minus four .. in other words the first 4-byte value stored on the stack.

Example: Assuming the the following values are stored at the indicated addresses and registers, evaluate the following operand specifiers:

Location:	0x100	0x104	0x108	0x10C	%eax	%ecx	%edx
Value:	0xFF	0xAB	0x13	0x11	0x100	0x1	0x3

1. %eax = 0x100
2. 0x104 = 0xAB
3. $0x108 = 0x108
4. (%eax) = Indirect - > 0x100 = 0xFF
5. 4(%eax) = Indirect -> 0x100 + 4 = 0x104 = 0xAB
6. 9(%eax, %edx) Indirect, Indexed = 9 + (0x100 + 0x3) = 9 + (0x103) = (0x10C) = 0x11
7. 0XFC(,%ecx,4) Indexed, Scaled = 0xFC + (4 * 0x1) = 0xFC + 0x4 = 0x100- = 0xFF
8. (%eax,%edx,4) Indexed, Scaled = (0x100 + (4 * 0x3)) = (0x100 + 0xc) = 0x10C -> 0x11

Data Movement

Data movement is the most commond operation type you will encounter when writing and evaluationg assembly code. There are three *classes* of data movement operations we will examine. Each instruction class, is defined over data lengths (1 byte - b, 2-bytes - w, 4-bytes -l).

- **MOV**: the MOV class of instructions copy the source value to the destination location or register. Both operands of a MOV instruction may not both be memory locations.
  Example: movl $99,%eax // stores the value 99 in register %eax
- **PUSH**: the PUSH class of instructions places a value at the top of the stack and then decrements the stack pointer.
  Example: pushl (%ecx) // stores the value in memory at the location stored in %ecx at the top of the stack.
- **POP**: the POP class of instructions, copies the value at the top of the stack into the memory address or register designated by the operand and incremenets the stack pointer.
  Example:popl %eax // pops a double-word from the stack and stores it in %eax.

Example: Given the following C, function write IA 32 Assembly for retrieving function parameter xp and lines 3 though 5 of the C code
C:

```
1       int swap_em(int *xp, int y) {
2       {
3               int x = *xp;
4
5               *xp = x;
6               return x;
7       }
```

IA-32 ASM:

```
movl    8(%ebp), %edx   // store *xp in %edx
movl    (%edx), %eax    // x = *xp
movl    12(%ebp), %ecx  // get y
movl    %ecx, (%edx)    // *xp=y
```