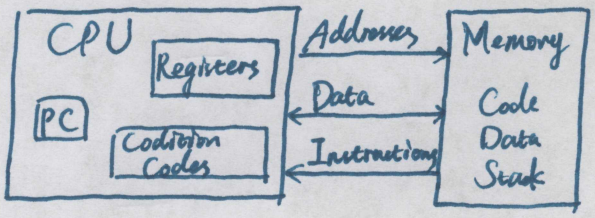
machine code { object code, the binary form (series of bytes) ; assembly code, the text form (target of compiler) } one to one mapping

machine code is intermediate form between how to write programs and how programs are actually executed on a machine

2004, intel's first 64-bit x86 processor, refered to as x86-64
the 64-bit machine can also run 32 bit code

Rather than giving an address from 0 to n-1, you actually give them by name to make use of registers, starting with %.

Assembly / Machine Code View



Cache is not visible in this graph. because it just automatically load recent staff. The only difference is if access that memory it would go faster than it would if it hasn't been cached. It's not visible in terms of there is no instructions to manipulate the cache, no way to directly access the cache.

Even if you run an executable program, there is actually some libraries get imported dynamically when the program first begins.

gcc -Og -S filename.c

-Og: optimization
if no specify, will generate tedious code
if -O1, turn on the optimizer
gcc does a lot of optimization
hard to understand "g" for debugging

-S: only run preprocess and compilation steps
C to assembly code

make to code readable
without being extravagant even rewrite the whole program

Assembly Characteristics: in integer data types, they don't distinguish sign vs. unsign in how it gets stored

addresses/pointers are just stored as numbers in computer, doesn't have any special significance

arrays, structs don't exist in the machine level, they are constructed artificially by the compiler

for x86, some instruction is as short as one byte, some can be as long as 15 bytes in the encoding, each instruction typically does one thing

Disassembler: reverse engineer from object code back to assembly code

all names in source code are completely lost at the assembly code level or the machine code level. it doesn't have access or understanding to the source code or assembly code, it figures out just by bytes in the object code file

objdump -d executable_filename

GDB: can make use of source code if available, but can also be used on object code which no source available.

disassemble function_name

application runs on computer
→ executable file → bytes that encode instructions

%e registers can hold 32 bits, %r registers can hold 64 bits

both of them are used (%e for ints, %r for long ints) in X86-64

%e is just the loworder 32 bits of %r entity

named locations where you can store values or retrieve values from them

we have to explicitly name registers to see where things should go and where things should come out of, not like a memory location by number addresses

X86-64 integer registers

specific purpose register: | %rsp | %esp | all others are usable for holding program data

stack pointer

arguments of function always come in some specific registers

%rdi 1st argument

%rsi 2nd argument ... can up to 6 of these before function execution

within the code, compiler will assign available registers to hold temporary data

complete memory addressing modes

D(Rb, Ri, S) Mem[Reg[Rb]+S*Reg[Ri]+D]

leaq Src, Dst : load effective address does the address computation, then writes that address, not the memory value, directly to the register

Dst : must be register

registers are not part of memory, neither part of cache

machine-level programmers have visibility of 16 registers, and memory

special one: %rsp stack pointer, where the state is manages procedures

%rip (%eip in IA32) instruction pointer, contains address of current executing instruction. Not a register accessed in normal way, though there are tricks to figure out its value → where are you in the program and what part program is being executed.

8 condition codes (1 bit flag), typically being set as side effect of other operations take place

CF, ZF, SF, OF, etc You can extract and get condition values directly, but it's very unusual. Common usage: read it and set 1 bit flag based on result in some other register or do conditional branch

condition codes explicit setting: compare and test

compare: two values compare with each other

test: one value, 0 or negative (two arguments of testq are the same)

Reading condition codes: setle 2's complement comparison also consider the most recent operation causes overflow or not

if the Xor of the sign and overflow flags is 1, the subtraction result is less than 0, otherwise is zero or greater

ZF | (OF^SF)

x==y? x-y<0? or overflow?

TAdd 只有同号才会 overflow

OF	SF	OF^SF	explanation
0	0	0	when $x-y \geq 0$ so $x<y$ is false
0	1	1	when $x-y<0$, $x<y$ is true
1	0	1	negative overflow, which means x与-y同为负y>0, $x<y$ is true
1	1	0	positive overflow, which means x与-y同为正, $y<0$, $x<y$ is false

for every 16 registers, you can directly set the lowest order byte of it to 0 or 1, and it doesn't affect other 7 bytes in the register

```
int gt(long x, long y) {      %rdi argument x
    return x > y;             %rsi argument y
}                             %rax return value

cmpq %rsi %rdi    # cmpq Src2, Src1, which essentially compare x:y (Src1 Src2)
                  cmpq compare instruction set 4 condition code registers
setq %al          # set when > set single byte register (lowest byte of %rax to either 0 or 1)
movzbl %al %eax     the other 7 bytes have whatever happen to be there before
```

movzbl move zero-extended byte to long
copy a byte from any place, like lowest order byte of some other register into a new register, and put zeroes to the left of it. That's zero-extension

but why %eax not %rax? x86-64: for any computation where the result is a 32-bit result, it will add zeros to the remaining 32 bits of the register
The byte-level operation only affect bytes, two-byte operation only affect two bytes, but four-byte instructions set upper bytes to zero.

in assembly code, if a name starts with colon, it's called a label, which is just a tag only visible in assembly code, not in the object code
it will be used to compute a address for determine where the jump will actually land

in assembly code, don't need to do anything special to return the value back from a function. just stick with the register %rax. the calling function will know where to find to return value

conditional move: optimization. compute both then and else expressions, only after that choose which one to use, makes the execution more efficient if they are fairly ~~common~~ simple compution (safe and no side effects)
modern processor goes along in a certain direction, really hard to get it stop or turn direction. Pipelining: start bringing in parts of one instruction before they're done with the next, go up to depth 20 or more instructions deep
fetching forward and future instructions while finishing up the one still remains
when hit a branch, do branch prediction. Mis-predicted branches are expensive, (stop, back up, turn the other way, start again) may take up to 40 cycles while conditional move is never very expensive. for example, for absolute value, compute both, move the value into a register in the last minute, which doesn't require stoping the whole processor and making turn

trick in Switch Statement Example:
ja: jump above, unsigned comparison (if legal values include negative values add bias so the first case is value 0)
if a number is negative, using unsigned comparison will make it a large positive value
jump if x>6, also x<0

break statements ⟹ ret
compiler generates the jump table, the details are filled with the assembler
if there are only two cases, 1 and 1,000,000, then the compiler will turn it into if-else
knows all values in advance compiler setup a tree of if-elses so only need log n number of tests