Assembly language x86 – gnu assembly (AT&T)

X86 is an RSA: instruction set architecture. Defines: 1. A set of instructions, 2. A way to pass operands to instructions, 3. How program access mem and regs. X86 is developed by intel and AMD, most common, especially server.
CISC: complex instruction set computer; **RSIC:** reduced instruction set computer: instruction do one thing, each instruction is 32 bits, 1 cycle.
ARM: another RSA, developed by acorn holdings, using RISC; most on mobile. Used by apple, qualcomm, lower power consumption

ARM and X86 are RSA, x86 use CISC, ARM use RISC; ARM lower power consum

`<OP> <src1> <src2> <dst>`
B – byte (8 bits), s = short (16), w = word (16), l = long (32), q = quad (64)
`$1, $num`: constant 1, num;
Prefix registers: `%E`: lower 32, `%R`: all 64 register, `%`: lower 16 bits
registers: ax: accumulator register (return), **bx:** base register, **cx:** counter, **dx:** data register, **SP:** stack pointer, **BP:** points to the base of stack frame, **R*n*:** (n = 8…15) general purpose pointers, **SI:** source index for string operations, **DI:** destination index of string, **IP:** instruction pointer, **FLAGS:** condition codes
`Var_4h, var_kh`: where k is number, an affect of -4 from pointer
`-4(%ebp)`: offset -4 from base pointer, or R[ebp] – 4

If a dst, src is enclosed by (): (src): dereference src

Instruction	Meaning
MOV dst, src	Copy data from src to dst
ADD dst, src	Add src to dst, store in dst
SUB dst, src	Subtract src from dst
MUL src	Multiply AX by src, result in DX:AX
DIV src	Divide DX:AX by src, quotient in AX
INC dst	Increment dst by 1
DEC dst	Decrement dst by 1
CMP dst, src	Compare dst and src, sets flags (SD = src - dst)
JMP label	Unconditional jump to label
JE label	Jump if equal (src – dst == 0)
JNE label	Jump if not equal (src - dst != 0)
JG label	Jump if greater (src - dst > 0)
JL label	Jump if less (src - dst < 0)
CALL label	Call subroutine at label
RET	Return from subroutine
PUSH src	Push src onto the stack
POP dst	Pop value from stack into dst
AND dst, src	Bitwise AND dst with src
OR dst, src	Bitwise OR dst with src
XOR dst, src	Bitwise XOR dst with src
NOT dst	Bitwise NOT (invert bits) of dst
SHL dst, n	Shift dst left by n bits (2^n)
SHR dst, n	Shift dst right by n bits (2^n)
NOP	No operation (does nothing)
HLT	Halt execution
endbr64	**Marks valid indirect branch targets (CET), sec.**
pushq src	Push 64-bit value onto the stack
cmpl dst, src	Compare 32-bit dst - src, sets flags
jle label	Jump if less than or equal (ZF=1 or SF???OF)
addl dst, src	Add 32-bit src to dst, store in dst
popq dst	Pop 64-bit value from stack into dst
retq	Return from subroutine in 64-bit mode
Movl src dst	Store long (4 byte) value @src to @dst in memory
Objdump -d xx.o	Disassemble object file to show you the assembly. Xx.o is the object file
Testl src, dst	Bitwise AND. Test reg, reg == 0 -> reg = 0; > 0 -> reg > 0; <0 -> reg < 0
Cmov(g,l,e…) src, dst	Conditional move if the previous test check is valid (testl src, dst)

base index scaled addressing mode in x86:
$$displacement(base\ register, index\ register, scale)$$
Effective address calculation:
$$effective\ address = m + (R[base] + R[index] * scale)$$

Effective address	What it means
M(%eax, %ebx, 2^k)	M + (R[eax] + R[%ebx] * 2^k)
N(%eax)	Mem[n + R[eax]]
(%eax)	Mem[R[eax]]

Eg: 4(%eax, %ebx, 2) = 4 + (R[%eax] + R[%ebx] * 2)
Eg: suppose y is in %rax, x is in %rbx, what instructions imple. y = *x?

```
Movq 0(%rbx), %rcx
Movl 0(%rcx), %ecx
Movl %ecx, 0(%rax)
```

Assembly derivative: `.def_name = value`, NOT instruction
Label: label, in form: `.label:`, NOT instruction
Load effective Address (LEA): preferred, computes address but **not** access it

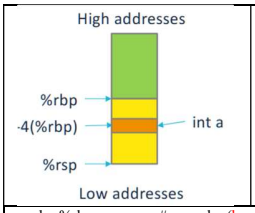
Eg: leal 2 (%rdi, %rax, 2), %eax ==== %eax = %rdi + %rax * 2 + 2
Eg: suppose %r1 = 1000, %r2 = 200, instr.: movl a(%r1, %r2, 4), %eax
Effective address: 8 + 1000 + 200*4 = 1808
Mem. Loads: 1 (only loads 1 time: fetch from mem addr. 1808)
Mem. Stores: 0 (it doesn't store to memory, but assign it to register %eax)

Abstraction CS: simplify complex sys. By hiding lower level details.
Control flow graph: CFG. A potential flow of exec. between basic blocks

```
int foo(int a) {
    int s = 0;
    for (int i = 0; i < 10; i++) {
        s += 1;
    }
    return 0;
}
```

How many basic blocks? - 4
1. int s = 0, int I = 0
2. if I < 10, goto 3, else go to 4
3. s += 1, i++, jump back to 2
4. return 0

control flow: cmp src - dst : check value of dst – src and set flag to jump
call graphs: shows which function call which function. Dynamically constru.
Function signature: func name + parameters + return type
C++ name mangling: take the datatype encoding of the function parameter in to the assembly encoded function name. res = parameter encoding + func name
stack management: high address on top, low address on bottom



%rsp – pointer top stack
%rbp – pointer base stack
Push, pop copu register to and from stack and adjust %rsp
Push writes to mem
Pop reads from mem

-4(%rbp) – refer to the local variable

```
push   %rbp            # save rbp (base pointer)
mov    %rsp,%rbp       # move stack pointer into the base pointer
mov    %rdi,-0x18(%rbp)   # place argument arr into -0x18(%rbp)
mov    %esi,-0x1c(%rbp)   # place argument length into -0x1c(%rbp)
movl   $0x0,-0x8(%rbp)    # place 0 into local variable sum
movl   $0x0,-0x4(%rbp)    # place 0 into local variable i
jmp    40059e <name_me+0x38> # unconditional jump to 40059e (loop condition check)

40059b:
mov    -0x4(%rbp),%eax    # move local variable i into %eax
cltq                      # convert long to quad (sign extend %eax)
lea    0(,%rax,4),%rdx    # load address (i * 4) into %rdx
                          # scale i by 4 for int array indexing
mov    -0x18(%rbp),%rax   # move arr into %rax
add    %rdx,%rax          # add %rdx and %rax into %rax (arr + i)
mov    (%rax),%eax        # place arr[i] into %eax
add    %eax,-0x8(%rbp)    # add arr[i] and sum, place result into sum
addl   $0x1,-0x4(%rbp)    # increment i by 1

40059e:
mov    -0x4(%rbp),%eax    # move i into eax
cmp    -0x1c(%rbp),%eax   # compare i and length
jl     400581 <name_me+0x1b> # conditionally jump to 400581 if i < len

mov    -0x8(%rbp),%eax    # move sum into %eax (return sum)
pop    %rbp               # restore %rbp
retq                      # return
```

Precomputation – Memoization, parallel

Multiple cores favor parallel programming for speed ups

```
# to count the number of 1s in a 64 bit binary number:
Extern "C" uint _popcount8(uint8_t t) {
   Uint count = 0;
   While(t) { # early stop
      if (t & 0x01) count++;
      t = t >> 1;
   }
}
# then, we divide the 64 bits into multiple 8 bits subproblems
and solve them parallelly o(n), with 3 instructions per bit
```

Optimization – fewest line of instructions. Compiler would auto optimize same operation value. For example,

```
Int cse(uint64_t t, uint64_t * data) {
   If (data[t + 1]) # movq 8(%rsi, %rdi, 8), %rax
      return data[t+1]; # movl $1, %edx
   Return 1;
}
```

CS architecture history

Moore's law: transistor/cm2 doubles ~ every 2 years. Not valid anymore
Clock speed scaling: 2* clock speed -> 8* power, **(ends freq scaling 2000s)**
Processor focus: specialization, throughout (parallel tasking), energy efficiency
Throughout: high yield -> cons: imp. single CPU is faster; hard implement.
Specialization: nvidia, tensorflow (ai)

Metrics and performance

Latency = work/throughput: 0.1s = 1mb/10mb/s
Speed up = latency before change / latency after change; if < 1, change hurts
$$S_{tot} = \frac{1}{\left(\frac{x}{S} + (1 - x)\right)}$$
Where x is the fraction of execution time of optimization, S is speedup optimization provides for that portion. Stot is total speedup
Metrics: latency * system weight, throughput * energy consumption, energy consumption * latency, work done / energy consumption
Example: for mobile devices, small delay and longer battery life, EDP small
For long dist. Net: Short transmit time, high throughput, longer dist. $m = \frac{t \times r}{d}$
No universal correct metric, prioritize latency over throughput
Amdahl's law: more widely applicable an optimization is, more valuable
$$execution\ time\ after\ improve = \frac{exe.time\ affected}{amount\ of\ improve.} + unaffected$$

Example: if old time is 200 hrs, 20% exec. on integer. How much faster is it needed to make code run 50 hours faster?
$$\frac{200}{150} = \frac{1}{\frac{0.2}{S} + 0.8}, S = neg, so\ impossible$$

Example: if program is 90% parallelizable and runs 100 seconds in single core, then what is ET if use 2 cores and no overhead?
$\frac{0.9}{2} + 0.1 = 0.55$, 0.55* 100 = 55, so the execution time is 55 sec

New latency = x*oldLatency / S + oldLatency * (1 - x)
Optimize common cases. As continue, common will become less common
➔ Focus on programs spent most time, make common case fast
Profiling: a tool that measures where program spend time. run with inputs, provides breakdown of time, which is x in Amdahl's law (`-pg`) (perf)
Flamegraph: call stacks, seq of func calls that led to current exe. Pt.
Big-O: constants are also important and overhead

Performance Equation

Performance equation: IC: instruction count, CPI: cycles per instruction, CT: cycle time (1/freq), ET: execute time. use performance counters
$$lantency = ET = IC * CPI * CT$$
Speed up: $S = ET_{before}/ET_{after}$
Improvement: IC: ISA designer, complier, programmer; **CPI:** complier, program language, algo. ISA (float point). **Cycle time:** hardware designer.
ISA: instruction set architecture: instruction format, operation, and mem inter.
Code: in C, we use Cfiddle (from cfiddle import *) to measure the code:

```
Start_measurement();
//code here
end_measurement();
return result;
```

measure if linear increase in time complexity: if input.len = n -> kn. measure IC, CPI, CT, ET at kn and calc $ratio = new/old$, if r = 1 then linear
Benchmark: standard program with standardized inputs, useful for computing Computer performance between systems, often specialize for particular purpose, subject to manipulation and create biased results, and a way to evaluate system performance; **used:** identify bottlenecks, repre. Typical work
processor power design: $P = V^2FaC + P_{idle}$, where V is processor core voltage, F is processor freq, a = activity factor, C is total capacitance of chip prop. to area. P_idle is base power consumption when doing nothing
since V and F are linearly related: $P \sim F^3aC + P_{idle}$, therefore, increase frq greatly increase power, and cause the **clock speed** to stop increasing (@2000s)
energy: $energy = power * time, program\ energy = ET * P$
battery time: $battery\ life = (battery\ cap\ in\ J)/program\ energy$

Example: suppose optimization A speed up fp calc.by 3.5; optimization B speed up non-fp calc. by 1.25; 20% in program is FP calc. and FP calc. has CPI =9, non FP has CPI = 2.5, calc speed up for opt. A and B:
Since $ET = IC * CPI * CT$, IC, CT does not change, so $S = \frac{CPI_{before}}{CPI_{after}}$
$$CPI_{before} = 0.2 * 9 + 0.8 * 2.5 = 3.8$$
$\boldsymbol{CPI_{afterA} = 0.2 * 9/3.5 + 0.8 * 2.5 = 2.51, S = 3.8/2.51 = 1.51}$
$CPI_{afterB} = \mathbf{0.2} * 9 + \mathbf{0.8} * \mathbf{2.5}/1.25 = 3.4, S = 3.8/3.4 = 1.117$

Performance of machine:

Actual performance: affected by implementation details: when input size are small, or algorithm is O(1) or is already optimal; **factors:** sys perf. Clock speed, complier flags, limitations of optimization by Amdahl's law
Asymptotic performance: big-O, when the inputs are large
Basic block: a sequence of instructions that always exec together
Static instructions: total number of instructions, no matter if reachable
Dynamic instructions: ~ would be executed when input.size = n
MIPS: millions of instructions per sec (time)
MFLOPS: millions of flipflop per sec (memory)

Complier optimizations

`Gcc -O0:` no optimization, all variables on the stack, extra loads and stores;
`Gcc -O0/1/2/3:` more optimization strength as x increases, prefer: `-o3`
Register: instruct complier to store the variable as register for faster access.
Register assignment: instruct complier to move parameter to reg.
Constant propagation: calculate constant at compile time
Loop invariant code motion: move repeated code above the loop
Strength reduction: use a simpler operation. Eg: x* 2^n is left shift n bits
Function inlining: call function Is expensive: overhead: ret, pop parameters to registers. Critical for programs that calls many small functions. Reduce IC
Loop unrolling: loop add many overheads. For simple loop instruction that execute kn times, complier can execute n same instructions in k loops.

If the iter. Times is too large (exceed the max value of index), infinite loop, because complier thinks the max value of the datatype is still less than value

compiler limitations

aliases: pointers can point almost anywhere in memory. Complier cannot check memory. When two different pointers points to same mem. Aliases use `__restrict__` to promise variable is not aliased. WARN: if break promise, code will be wrong. By default. **Different datatype pointers point to different memory.** Use `T targetType = reinterpret_cast<T>(expression);` to convert one pointer type to another, even if the types are completely unrelated
function calls: function inlining (`inline` add func header) remove overhead
memory op is slow

C++ templates

Are static code generation mechanisms. Value set in complier time
For example: **array, struct**
Array access: use movl (%rdi, %rsi, 4), %eax, no add, 1 load and 1 store
Struct access: movl 6(%rdi), %eax, no add, 1 load and 1 store

Inheritance

Class child : public parent