1. 데이터 이동 명령어 (ALU 미 사용)
   1. X86 – 64 vs IA32 : 한번에 다룰 수 있는 데이터 타입의 크기가 변화
      1. 어셈블리 명령어에 suffix ‘q’ (8 bytes) 추가
      2. 레지스터 개수 늘어남 (+8)
      3. 레지스터를 통한 오퍼랜드 전달 (RISC의 특성)
         1. 스택 연산이 필요없다.
2. 산술 연산 명령어 (ALU 사용)
   1. 종류 : 단항 연산, 이항 연산, 시프트 …
   2. 이항 연산 주의 : SRC는 연산 주체 DST는 피 연산자 ex) SUB S, D => D = D – S
   3. LEA (Load Effective Address)
      1. mov에서 사용하는 주소 연산을 실제 산술연산에 사용
      2. D(b, i , s) => b + I \* s + D
      3. 메모리 연산이 아니라 산술 연산이다
   4. 효율적인 활용
      1. 인자는 (8+4n)(%ebp)형식으로
      2. 연산 순서는 컴파일러가 알아서 최적화
      3. 가능하면 lea과 sal, sar 을 사용
      4. 상수끼리의 연산은 미리 계산해서 하나의 상수로 만듬
      5. 곱셈의 오버플로우 : 단항 연산 mul을 사용하여 64비트처럼 사용할 수 있다. (미리 레지스터를 예약해서 사용하는 방식)
3. 제어문
   1. 구현 방식
      1. 조건 코드
      2. 점프 명령어 (pc값 변경)
   2. 프로세스의 상태
      1. 프로세스 정보를 레지스터에 저장
      2. 임시데이터, 스택 위치(%esp), 제어위치(PC), 조건 코드 (Condition Code))
   3. IF 의 구현
      1. 불린 연산을 Condition Code에 저장한 뒤 그 결과에 따라 Jump 연산 수행
   4. 조건 코드
      1. 비트 레지스터 (플래그 저장)
         1. Carry, Zero, Sign, Overflow FLAGS…
         2. 산술연산에 의해 자동적으로 값이 설정됨 (lea 제외: ALU안씀)
         3. 조건에 따른 적당한 산술연산을 수행한 뒤에 생성되는 조건 코드를 참조함.
         4. CF : unsigned 수준의 overflow, OF : signed 수준의 overflow (2의 보수연산 부적절)
      2. Compare & Test
         1. 비교(뺄셈), Equal(&) 연산 처리 후 결과를 조건 코드에 저장
         2. 변수 레지스터 값을 변경시키지 않는다는 점이 장점.
      3. 조건 코드 활용
         1. 복잡한 조건을 확인하기 위해서 조건 코드의 조합이 필요(set)
            1. Set + suffix : 접미사는 조건 코드 4개 비트의 해석방식을 규정
            2. E(==)
            3. comp %eax, %edx //a와 d를 비교
            4. setg %al //뒤에 것이 앞에 것(순서 중요)보다 크냐? 를 al에 저장
            5. movzbl %al, %eax //eax의 나머지 칸에 0으로 비워서 제대로된 값을 넣어줌 (%eax는 암묵적 리턴 변수)
         2. 조건 코드 값에 따라 실행 제어 이동 (jump)
            1. Pc값 변경 jmpl (%eax) == movl (%eax, %eip(pc레지스터))
            2. 무조건 점프 조건 신경 안 씀.

Ex) 반복문의 끝에서 다시 조건체크로 넘어가는 부분

직접 이동 위치(라벨)를 정할 수 있고, 오퍼랜드에서 받아올 수 있음

Jmp Label , jmp \*%eax

점프 위치는 상대좌표로 주어져야 한다. 실제 어떤 주소에 로드될지 알 수 없기 때문

점프 위치는 현재 PC값과의 차이(0ffset), PC값은 현재 수행하는 라인 바로 뒤라는 것에 주의

어셈블리는 <함수명 + offset>으로 적어준다. 읽기 쉬움

* + - * 1. 조건 점프

Suffix 를 붙여서 특정 조건하에 점프 수행하도록

컴파일러는 일반적으로 Else의 조건으로 체크하도록 코드를 짠다.

Else가 없는 경우 jmp명령어 개수를 줄이기 위해서

* + - 1. 조건 코드 값에 따라 데이터 이동 (move)
         1. 점프를 안쓰려고 만든 것 점프를 안 쓰면 성능에 큰 도움이 된다.
         2. CPU 파이프라인 때문 (제어 – ALU – 레지스터)
         3. 미리 다음 연산을 준비하고 있음
         4. 하지만 jump를 사용하면 다음 연산에 대해 준비 할 수 없으므로 PipeLine이 끊어진다.