시스템 아키텍처 요약

Machine Level Programming

141019 김연우

1. 어셈블리 코드의 개념
   1. 어셈블리란?
      1. ISA(Instruction set achitechure)를 사람이 알기 쉬운 형태로 전달하는 것
      2. 기계어 & 어셈블리
         1. 기계어: 컴퓨터가 수행할 작업을 인코딩한 연속된 바이트 파일
         2. 어셈블리: 기계어 코드를 사람이 읽기 쉬운 형태로 표시한 것
            1. 고급언어보다 하드웨어의 동작을 구체적으로 유도할 수 있다.
            2. 하지만 짜기 어렵다. 요즘 컴파일러 >>>>>> 나
            3. 근데 왜 공부?

컴파일러의 동작을 이해하여 코드의 비효율성을 잘 분석하고,

메모리를 효율적으로 사용함으로써 프로그램 성능을 최적화 하는데 유리하다.

프로그램의 에러를 효율적으로 발견하고 제거할 수 있다.

소프트웨어 보안 약점을 찾고 회피하는데 효과적이다.

* 1. 인텔 프로세서
     1. x86 (ISA 명령어 셋 아키텍처의 이름, IA32랑 거의 같다고 생각하자)
     2. 8086(1978년 프로세서)부터 호환성을 유지하면서 확장했음
     3. CISC 구조(복잡한 명령어 셋, (cf) RISC: 단순한 명령어 셋))의 대표주자.
     4. 클럭 구조를 사용하여 최적화
     5. 컴퓨터 아키텍처의 발전과정은 명령어 셋 아키텍처와 밀접한 연관이 있다.
        1. 아키텍처는 설계도 : ISA
        2. 마이크로 아키텍처가 구체적인 하드웨어의 구성을 말함.
  2. C vs 어셈블리
     1. CPU란 결국 상태기계
        1. 처리한 결과 레지스터에 저장
        2. 명령어 하나를 처리하고 PC값에 따라 다음 명령 수행
        3. 현재 상태에 따라 조건 레지스터가 다음 상태를 결정(?)
     2. c코드를 컴파일 하면 어셈블리 코드로 변경된다. (gcc -01 –S cosde.c)
     3. 어셈블리 코드가 어셈블러를 거쳐 바이너리 코드(기계어)로 변경된다.
        1. CISC는 기계어의 명령 단위가 서로 다르다. (RISC는 똑같음)
        2. 다른 코드와의 연결정보(LINK)는 누락된 상태.
        3. 기계어 표현은 사용이 제한이 있다. (2항연산만 가능…)
     4. 기계어로부터 역 어셈블이 가능하다.(objdump –d prog)
        1. 목적 코드를 분석가능 (reverse engineering)
        2. 링크하기 전까지 함수나, 변수의 주소는 결정되지 않는다.
        3. 디버거를 사용해서 역어셈블도 가능
           1. 리눅스 디버거 : gdb dissamsemble….
           2. 비주얼 스튜디오에서 가능 : break걸고 디버그 창\디스어셈블리

어셈블리 코드 형식이 다름 AT&T(리눅스) vs 인텔(윈도우)

* + 1. 링커는 다른 목적파일들을 연결하고 변수 ,함수등의 데이터들의 실제 주소를 결정한다.
  1. 어셈블리 기본
     1. 정수 1, 2, 4, 바이트, float 2, 4, 8(?)
     2. WORD 2바이트 ( 16비트 호환성), DWORD…QWORD
     3. ISA에는 메모리 단위별로 명령어세트를 따로 가지고 있다.
        1. Ex) add(l) : 32 , add(b) : 8 , add(w) : 16, add(q) : 64
     4. 명령어 종류
        1. 데이터 이동 : 메모리 로드 세이브 카피
        2. 산술 논리 연산 : 메모리(레지스터)에 있는 데이터 조작
        3. 제어 이동 : 조건부 분기, 함수 호출

1. 어셈블리 코드의 형식
   1. IA32 정수 레지스터
      1. 레지스터 사이즈는 컴퓨터 비트 사이즈 32-64 bits
      2. 16bit짜리 가상 레지스터를 쓸 수 있다. (high low로 나눠서 사용하기도)
      3. 쓸 수 있는 저장용 레지스터 8개
      4. 범용 레지스터 6개 변수 담는 용도
         1. %eax (리턴값), %ecx (counter), %edx(data), %ebx(base : 변수에 해당하는 주소를 저장), %esi (source index) , %edi (dst index) 는 훼이크고 그냥 막 쓴다. 하지만 가능하면 용도에 맞게 쓸려고 한다.
      5. ??? : %esp 스택 포인터(?), %ebp 베이스 포인터(?)
   2. 명령어 형식
      1. Operator + Operand
      2. Operand 형식
         1. 레지스터 : %eax
         2. 상수 정수 데이터 : $0x400 (상수는 앞에 $를 쓴다.)
         3. 메모리 : (%eax) 괄호 붙이거나 그냥 숫자 ($이 없다\*)
      3. 문법
         1. OP + Src + Dst : movl %eax ,%edx
         2. 문법이 정해져 있음. (RISC는 간결(일반 연산은 모두 레지스터, 메모리 이동할 때에만 메모리 사용), SISC는 더러움)
            1. Ex) Move 는 src + dst 이때 src가 메모리이면 dst는 레지스터만 가능
   3. 메모리 주소지정 모드
      1. 일반 지정 : Movl (%ecx), %eax;
      2. 변위 지정 : Movl 8(%ebp), %edx 상수 D는 참조할 변수의 변위를 표시
      3. Generic 변위지정 : 배열이나 구조체 클래스의 단위로 접근하기 위한 메모리 표현방법
         1. D(Rb, Ri, S) => movl 0x2048(%ebp, %edi , 4)
         2. Rb(head의 위치) + (Ri \* S(한 단위의 크기: 1,2,4,8 only)) + D(상수 변위)
         3. D에 큰 단위의 위치 값을 넣고 Rb에 그 내부의 단위에서 Base를 지정해주는 방식을 사용할 수 있다.
2. 데이터 이동 명령어
   1. MOV 일반 이동, MOVS 부호비트 고려 확장 MOVZ 그냥 0으로 채우는 확장
   2. PUSH ,POP ??? 스택!
      1. 스택 : 함수의 호출을 지원하고 지역변수를 저장하는 영역, CPU가 지원함
         1. %esp에 스택의 top주소를 유지함.
            1. 함부로 %esp에 값을 넣으면 Stack영역 터짐
         2. PUSH %eax == SUB $4 %esp MOV %eax, (%esp)
         3. 자주 쓰이므로 한번의 연산으로 해결
   3. %ebp ?? Base Pointer , Stack Frame
      1. 함수가 사용하는 영역을 표시하는 것
      2. Esp는 전체 프로그램의 스택의 top
      3. Ebp는 현재 진행되는 함수의 진행 위치 등을 표현
   4. 스택을 통한 함수 구현
      1. Setup : pushl %ebp, movl %esp, %ebp
         1. 함수 호출 직전의 %ebp레지스터 값을 스택에 저장
         2. 이 결과 변경된 스택 포인터 값을 %ebp에 저장
      2. Body : ~~
      3. Finish : popl %eax, popl %ebp
         1. Setup 부분에서 스택에 저장한 이전 %ebp값을 읽어와서 이전 위치로 복구
      4. 각 함수가 호출될 때 해당 함수가 사용할 스택의 기준 주소를 %ebp에 저장해서
      5. 스택 포인터 값이 변해도 기준이 되는 주소를 유지하기 위함.
3. 산술 연산 명령어
   1. 함수가 호출되면 스택의 top에 인자, 리턴 주소가 자동으로 쌓인다.
   2. 일반적으로 시작부분에 이전 진행상황을 저장해서 이번 호출된 함수가 종료된 이후에 팝해서 다시 원래의 진행상황으로 돌아가게 한다.
4. 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.
5. 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)
      6. 예제



1. 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 코드 레지스터에 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)
            6. 예제



* + - 1. 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.

1. 어셈블리 조건문 연계
   1. 순환문 (loop)
      1. 조건부 점프 명령어. 점프 주소가 음수 값 (앞으로 가는 점프 문)
      2. 컴파일러는 Do – While 형태로 순환 문을 해석한다.
      3. 일반 While문이면, 맨 앞에 조건부 점프 문 하나 넣으면 됨.
      4. 진입 시점에서는 False인 경우를 점프. 반복 시점에서는 True인 경우 점프
      5. For문의 경우 일반 While문과 크게 다르지 않다. 초기값 세팅
      6. %eax는 리턴값 저장용
      7. 예제





Int foo(int a)

{

Int I;

Int result = a + 2;

For( I =0 ; I < a ; ++i)

{

Result += I + 5;

Result \*= (I + 3);

}

Return result;

}

* 1. Switch Case
     1. 점프 테이블 활용
     2. 점프 주소 배열을 이용하는 간접 점프 (포인터 배열: jmp \*Label(, %eax, 4))
     3. Case의 라벨은 배열의 인덱스로 변환하므로 값이 조밀하게 분포하는 경우에 적합
     4. 점프 테이블은

1. 프로시저 (함수)
   1. 스택
      1. 스택의 존재 이유
         1. CPU는 그저 연속된 어셈블리 코드를 처리할 뿐, 프로시저 단위를 알지 못함
         2. 프로시저는 인간이 코드들을 효율적 구조적으로 활용하기 위해 만들어진 것
         3. 독립적 구조로서 프로시저는 분리된 컨텍스트를 갖고 있어야 함. (컨텍스트 스위칭!)
         4. 함수가 호출(점프)이 되면, 호출된 함수의 컨텍스트를 사용, 리턴되면 본래의 컨텍스트로 돌아옴
         5. 이런 컨텍스트가 저장되는 공간이 스택.
      2. 스택의 구조
         1. 스택의 top 포인터(%esp)
         2. pushl, popl 명령어로 스택에 save/load & 스택 포인터 유지
         3. 스택 프레임 : 하나의 프로시저에 할당된 부분
         4. 스택 프레임(의 시작주소) 포인터(%ebp)
         5. 함수 호출/반환할 때 스택 레지스터 변경/저장
         6. 기존에 사용하던 레지스터 정보는 새로운 프레임의 상단에 Savaed Register에 저장한다.
         7. 프로시저(함수)의 인자는 호출한 쪽의 스택 프레임이 가지고 있다.
      3. 스택 활용 방식
         1. 호출하는 프로시저는, 뒤쪽 인자부터 저장. 맨 마지막에 리턴 주소저장.
         2. 호출된 프로시저는, 기존 프레임 포인터 값 저장.
         3. 새로운 명령어 : call, ret , leave
         4. Call : 현재 스택에 %eip (pc) 값을 저장, & esp ++;
         5. Ret : 저장된 이전 pc값을 pop해서 pc값 변경. (점프 노필요)
         6. Leave : 리턴하기 전에 리턴 값을 리턴 어드레스에 저장해둠
         7. Caller-Save, Callee-Save : 프로시저 호출 전에 레지스터에 저장된 값을 호출된 프로시저가 리턴한 후에도 계속 이용하려고 할 때, 호출된 프로시저에서도 동일한 레지스터를 사용할 가능성이 있기 때문에 메모리에 저장했다가 복구해야 한다. 이때 프로시저 호출 전 호출자가 프로시저가 저장하는 것으로 약속한 레지스터를 Caller-Save 레지스터라 하고, 호출된 프로시저 내에 저장하는 것으로 약속된 레지스터를 Callee-Save 레지스터라고 한다.
         8. 예제





1. 프로세서
   1. 명령어 집합
      1. RISC : 빠르게 실행할 수 있는 단순한 형식의 명령어들, 명령어 길이가 같고 적은 개수의 명령어로 집합 구성
      2. CISC : 비교적 복잡한 계산도 하나의 명령어로 제공, 각 명령어 수행시간 길다.
   2. 프로세서 실행단계
      1. Fetch : Instruction 메모리에서 명령어 읽기
      2. Decode : 프로그램 레지스터에서 읽기
      3. Execute : 산술/논리 연산 혹은 주소계산
      4. Memory: 데이터의 읽기 혹은 쓰기
      5. Write Back : 프로그램 레지스터에 쓰기
      6. PC update : 다음 명령어의 주소로 PC갱신
   3. 파이프 라인
      1. 파이프 라인 해저드 : 프로세서에 실행하는 명령어들 사이의 의존관계에 의해 파이프 라인 실행을 진행할 수 없는 상황.
      2. 파이프 라인의 장점
         1. 단위 시간 동안 실행된 명령어 개수인 시스템 처리율이 증가.
         2. 명령어 하나의 실행 완료시간인 응답 시간이 증가.
         3. 안정화 되었을 때, 한 클럭에 프로세스 하나가 종료됨.
   4. 암달의 법칙 :
      1. 시스템 일부분의 성능을 개선했을 때, 전체 시스템의 성능 개선 비율은 해당 부분이 차지하는 비중에 따라 효과가 나타난다.
      2. 전체 시스템의 성능을 효과적으로 개선하기 위해서, 시스템에서 차지하는 비중이 큰 부분의 성능을 개선해야 한다.
   5. 프로 파일링
      1. 프로그램을 구성하는 각 함수의 호출 횟수와 실행 시간, 비율 등을 알 수 있다.
      2. 이 결과에 따라 프로그램에서 자주 사용되는 함수와 실행 시간의 병목이 되는 부분을 식별할 수 있고, 암달의 법칙에 따라 이런 부분의 최적화에 집중하여 전체 프로그램의 성능 개선을 효과적으로 수행할 수 있다.
   6. 벤치마크
      1. 컴퓨터 시스템의 성능을 측정하기 위해 선택된 프로그램 집합, 실제 프로그램에서 추출해 내거나 인위적으로 생성한 워크로드를 실행하는 프로그램