

NA SPOKOJNIE

czwartek, 20 czerwca 2024 09:11



Skoki nazywamy też rozgałęzieniami.

Nasze skomplikowane warunki musimy zapisywać za pomocą warunków prostszych.

Tomczak skokach ze śladem służących np. do tworzenia funkcji. Skoki ze śladem zapisują miejsce, z którego skaczą.

Tomczak powtarza, że nauka we własnym tempie jest ważna. "Musicie państwo na spokojnie"

Tomczak porównuje naukę WDWK do gry na gitarze. "Patrząc na pianistę nie nauczycie się grać, musicie ćwiczyć"

W szybkim kodzie należy unikać skoków jeśli nie jest to konieczne! Mogą być bardzo kosztowne.

Tomczak wspomina instrukcje działające warunkowo z poprzedniego wykładu. Korzystając z nich nie zmieniamy adresu tylko sprawdzamy warunek i wykonujemy instrukcję lub nie.

Jeśli nie ma potrzebnej instrukcji: sztuczki! Przemnożenie wyniku przez jeden lub zero. Niekoniecznie arytmetyczne ale może np. logiczne

Wykonanie większej liczby operacji może trwać krócej niż jeden rozkaz skoku.

Instrukcje prefiksowane (w kartach graficznych):

Istnieje dodatkowy rejestr zawierający flagi bitowe i dany rozkaz lub sekwencja rozkazów może być poprzedzona warunkiem, który mówi czy je wykonać czy nie. Wszystkie te rozkazy są jednak pobierane i dekodowane, ale w zależności od tego mogą nie być wykonane. Są pomijane.

Unikanie rozgałęzień

instrukcje wykonywane warunkowo (IA-32)

cmov <i>war</i> <i>arg</i> , <i>zmienna</i>	; if <i>war</i> = true then <i>arg</i> = <i>zmienna</i>
cmpxchg (<i>acc</i>), <i>zm1</i> , <i>zm2</i>	; if <i>acc</i> = <i>zm1</i> then <i>zm1</i> = <i>zm2</i> ,
	; <i>acc</i> = <i>zm1</i>

alternatywne podstawienie

if *warunek* **then** *X*=*A* **else** *X*=*B*

TRUE=00...01,	TRUE=11...11,	
FALSE=00...00	FALSE=00...00	
sub <i>X</i> , <i>A</i> , <i>B</i>	sub <i>X</i> , <i>A</i> , <i>B</i>	<i>X</i> := <i>A</i> − <i>B</i>
...	...	

FALSE=00...00	FALSE=00...00	
sub X,A,B	sub X,A,B	X:=A-B
set not-war Z	setwar Z	
sub X,A,B	—	Z:= 11...11 if war
and X,X,Z	and X,X,Z	X:=(A-B)&Z
add X,X,B	add X,X,B	X:=B+Z&(A-B)

Tomczak odwołuje się do przykładu prof. Biernata.

Na obrazku wyżej jest wykonywana operacja iloczynu logicznego z samymi jedynkami lub z samymi zerami, wynik działania jest ustawiany albo na same zera albo na same jedynki w zależności od jakiegoś warunku. Potem przemnażamy jeden z wyników przez same jedynki. Dodatkowo sztuczka: TO CO MNOŻYMY PRZEZ JEDYNKI TO FRAGMENT WYRAŻENIA, KTÓRE OBLICZAMY.

Po dokładnej analizie komentarzy okaże się, że:

- Do b dodajemy sumę A-B i zostaje samo A lub
- Do b dodajemy zero i wtedy zostaje samo B

Czego Tomczak nie podkreśla to, że mamy tutaj składnię IA-32, dla której:

SUB X,A,B oznacza odjęcie B od A i zapisanie do X

SET not-war Z ustawienie z na 00000001 jeśli warunek nie zachodzi TO JEST NAJWAŻNIEJSZE!

WARUNKOWE WYCZYSZCZENIE OPERACJI A-B Z REJESTRU Z

Najważniejszy tak naprawdę jest ten set, bo to on ustawia Z na odpowiednią wartość, która potem jest przemnożona z wynikiem odejmowania:

and X, X, Z

, czyli odejmowanie może być wyzerowane przez co nie wpływa na dalsze obliczenie

add X, X, B

Podsumowując krótko jeśli jeszcze ktoś nie załapał :

Dwie pierwsze kolumny to dwa rozwiązania, trzecia to wyjaśnienie, ale ja mam własne (polecam zacząć od drugiej kolumny):

1. Pierwsza kolumna (DZIWNA METODA):

JEŚLI PRZYJMIEMY, ŻE PRZEZ set not-war Z rozumiemy negację TRUE, a nie FALSE

- X = A - B
- IF WARUNEK THEN (każdy bit będzie przeciwny) Z =11111110 ELSE Z=11111111, zaczniemy od tego, że to już jest dziwne, ale taka najwidoczniej jest ta architektura
- X = A-B # nie mam pojęcia po co drugi raz
WYDAJE MI SIĘ, ŻE TU JEST BŁĄD, LOGICZNE WYDAJE SIĘ, ŻE JEŚLI WARUNEK JEST PRAWDZIWY POWINNIŚMY NAPRAWIĆ TRUE BO Z 111...110 DALEKO NIE ZAJDZIEMY, TO GŁUPIE, ALE NIE MA INNEJ OPCJI JAK DODAC BIT WARTOŚĆ 1 DO Z
add Z, Z, 1 (Z = Z+1) i wtedy jeśli Z było prawdą zostanie PEŁNĄ PRAWDĄ JAKIEJ CHCEMY, a jeśli nie TO WSZYSTKIE BITY ZOSTANĄ WYZEROWANE PO DODANIU, A O TO NAM CHODZI
- X = X AND Z czyli zostawiamy wynik w X tylko jeśli warunek wystąpił
- X = X + B
- JEŚLI WARUNEK WYSTĄPIŁ X = A-B +B czyli X = A
- JEŚLI WARUNEK NIE WYSTĄPIŁ X = 0 + B czyli X = B

Ogólnie Tomczak wspomina, że raczej o to mu chodziło:

JĘŚLI PRZYJMIEMY, ŻE PRZEZ set not-war Z rozumiemy FALSE

1. $X = A - B$
2. IF WARUNEK THEN (każdy bit będzie przeciwny) $Z = 00000000$ ELSE $Z = 00000001$, zaczniemy od tego, że to już jest dziwne, ale taka najwidoczniej jest ta architektura
3. $X = A - B$ # nie mam pojęcia po co drugi raz
WYDAJE MI SIĘ, ŻE TU JEST BŁĄD, LOGICZNE WYDAJE SIĘ, ŻE JĘŚLI WARUNEK JEST PRAWDZIWY POWINNIŚMY NAPRAWIĆ OBA BO Z TAKIMI DALEKO NIE ZAJDZIEMY, TO GŁUPIE, ALE NIE MA INNEJ OPCJI JAK ODJĄĆ BIT WARTOŚĆ 1 DO Z
sub Z, Z, 1 ($Z = Z - 1$) i wtedy jeśli warunek był prawdą to Z zostanie PEŁNĄ PRAWDĄ JAKIEJ CHCEMY, a jeśli nie TO WSZYSTKIE TEN JEDEN BIT NA KOŃCU ZOSTANIE WYZEROWAN I BĘDZIEMY MIELI PIĘKNY ZEROWY FAŁSZ
4. $X = X \text{ AND } Z$ czyli zostawiamy wynik w X tylko jeśli warunek wystąpił
5. $X = X + B$
6. JĘŚLI WARUNEK WYSTĄPIŁ $X = A - B + B$ czyli $X = A$
7. JĘŚLI WARUNEK NIE WYSTĄPIŁ $X = 0 + B$ czyli $X = B$

2. Druga kolumna (PROSTA METODA):

1. $X = A - B$
2. IF WARUNEK THEN $Z = 11111111$ ELSE $Z = 00000000$
3. $X = X \text{ AND } Z$ czyli zostawiamy wynik w X tylko jeśli warunek wystąpił
4. $X = X + B$
5. JĘŚLI WARUNEK WYSTĄPIŁ $X = A - B + B$ czyli $X = A$
6. JĘŚLI WARUNEK NIE WYSTĄPIŁ $X = 0 + B$ czyli $X = B$

I W TEN SPOSÓB ZAGRALIŚMY BEETHOVENA

O skoku raz jeszcze

piątek, 21 czerwca 2024 21:10

Tomczak cofa się do skoków warunkowych.

Jcc—Jump if Condition Is Met

Opcode	Instruction	Description
77 <i>cb</i>	JA <i>rel8</i>	Jump short if above (CF=0 and ZF=0)
73 <i>cb</i>	JAE <i>rel8</i>	Jump short if above or equal (CF=0)
72 <i>cb</i>	JB <i>rel8</i>	Jump short if below (CF=1)
76 <i>cb</i>	JBE <i>rel8</i>	Jump short if below or equal (CF=1 or ZF=1)
72 <i>cb</i>	JC <i>rel8</i>	Jump short if carry (CF=1)
E3 <i>cb</i>	JCXZ <i>rel8</i>	Jump short if CX register is 0
E3 <i>cb</i>	JECXZ <i>rel8</i>	Jump short if ECX register is 0
74 <i>cb</i>	JE <i>rel8</i>	Jump short if equal (ZF=1)
7F <i>cb</i>	JG <i>rel8</i>	Jump short if greater (ZF=0 and SF=OF)
7D <i>cb</i>	JGE <i>rel8</i>	Jump short if greater or equal (SF=OF)
7C <i>cb</i>	JL <i>rel8</i>	Jump short if less (SF<>OF)
7E <i>cb</i>	JLE <i>rel8</i>	Jump short if less or equal (ZF=1 or SF<>OF)
76 <i>cb</i>	JNA <i>rel8</i>	Jump short if not above (CF=1 or ZF=1)
72 <i>cb</i>	JNAE <i>rel8</i>	Jump short if not above or equal (CF=1)
73 <i>cb</i>	JNB <i>rel8</i>	Jump short if not below (CF=0)
77 <i>cb</i>	JNBE <i>rel8</i>	Jump short if not below or equal (CF=0 and ZF=0)
73 <i>cb</i>	JNC <i>rel8</i>	Jump short if not carry (CF=0)
75 <i>cb</i>	JNE <i>rel8</i>	Jump short if not equal (ZF=0)
7E <i>cb</i>	JNG <i>rel8</i>	Jump short if not greater (ZF=1 or SF<>OF)
7C <i>cb</i>	JNGE <i>rel8</i>	Jump short if not greater or equal (SF<>OF)
7D <i>cb</i>	JNL <i>rel8</i>	Jump short if not less (SF=OF)
7F <i>cb</i>	JNLE <i>rel8</i>	Jump short if not less or equal (ZF=0 and SF=OF)
71 <i>cb</i>	JNO <i>rel8</i>	Jump short if not overflow (OF=0)
7B <i>cb</i>	JNP <i>rel8</i>	Jump short if not parity (PF=0)
79 <i>cb</i>	JNS <i>rel8</i>	Jump short if not sign (SF=0)
75 <i>cb</i>	JNZ <i>rel8</i>	Jump short if not zero (ZF=0)
70 <i>cb</i>	JO <i>rel8</i>	Jump short if overflow (OF=1)
7A <i>cb</i>	JP <i>rel8</i>	Jump short if parity (PF=1)
7A <i>cb</i>	JPE <i>rel8</i>	Jump short if parity even (PF=1)
7B <i>cb</i>	JPO <i>rel8</i>	Jump short if parity odd (PF=0)
78 <i>cb</i>	JS <i>rel8</i>	Jump short if sign (SF=1)
74 <i>cb</i>	JZ <i>rel8</i>	Jump short if zero (ZF = 1)
0F 87 <i>cw/cd</i>	JA <i>rel16/32</i>	Jump near if above (CF=0 and ZF=0)
0F 83 <i>cw/cd</i>	JAE <i>rel16/32</i>	Jump near if above or equal (CF=0)
0F 82 <i>cw/cd</i>	JB <i>rel16/32</i>	Jump near if below (CF=1)
0F 86 <i>cw/cd</i>	JBE <i>rel16/32</i>	Jump near if below or equal (CF=1 or ZF=1)
0F 82 <i>cw/cd</i>	JC <i>rel16/32</i>	Jump near if carry (CF=1)
0F 84 <i>cw/cd</i>	JE <i>rel16/32</i>	Jump near if equal (ZF=1)

Argumenty w skokach warunkowych są podawane na dwa sposoby:

1. Skok względny, wtedy to jest przesunięcie względem bieżącego adresu (najczęściej używany), jeśli używamy etykiety jako adresu skoku to assembler wylicza ile trzeba się przesunąć, gdyby to ręcznie wpisywać to jeśli zmodyfikujemy kod między tym miejscem, z którego skaczemy, a tym miejscem do którego skaczemy musielibyśmy modyfikować tę wartość.
2. Skok bezwzględny ("skok z adresem bezwzględnym" "skok pośredni"), argument będący argumentem instrukcji skoku zawiera adres miejsca, w którym jest zapisany adres docelowy skoku, tym miejscem może być albo rejestr albo komórka pamięci, z rejestru szybko, z pamięci wolno.

Uwaga! Może być to kosztowne lub bardzo kosztowne, bo jest nieprzewidywalne.

Musimy dokończyć wykonywanie bieżącego rozkazu zanim będziemy w stanie wykonać skok.

Tomczak nawiązuje do dokumentacji:

JMP

Near and Short Jumps.

- When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute offset (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register). A near jump to a relative offset of 8-bits (rel8) is referred to as a short jump. The CS register is not changed on near and short jumps.
- An absolute offset is specified indirectly in a general-purpose register or a memory location (r/m16 or r/m32). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.
- A relative offset (rel8, rel16, or rel32) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

To jest to o czym Tomczak tutaj rzekł i można sobie poczytać. Prof. Poleca.



ABI

- *ang. application binary interface*, interfejs binarny aplikacji
- **nie mylić z API**
- definiuje:
 - typy danych (rozmiar i ułożenie w pamięci)
 - sposób wywoływania funkcji (*ang. calling convention*): przekazywanie parametrów, zwracanie wartości, „właściciela” rejestrów, ramkę stosu
 - sposób wywoływania funkcji **systemowych**
 - formaty i nazwy plików bibliotecznych, nagłówkowych, programów, itd.

136

- I. ABI (Application Binary Interface) - Jeśli używamy Linuxa w 32 bitowej wersji to ABI opisuje sposoby konstruowania kodu i to jak powinien się zachowywać. Programy linuxowe są konstruowane mniej więcej zgodnie z tymi regułami.
(Ten rysunek wyżej będzie później, ale już teraz go wrzucam)

Figure 3-45: Branch Instruction, All Models

C	Assembly
<pre>label: . . . goto label;</pre>	<pre>.L01: . . . jmp .L01</pre>

Mamy kod w C i Assembly, ten sam skok. Argumentem rzeczywistym jmp będzie ujemna liczba w reprezentacji U2, to przesunięcie będzie wyliczone przez assembler na podstawie tego co będzie zamiast tych trzech kropek.

Uwaga! Jeśli podalibyśmy z palca kod, który cofnie nas w środek instrukcji program się zawiesi, bo procesor nie zrozumie o co biega.

C	Assembly
<pre>switch (j) { case 0: . . . case 2: . . . case 3: . . . default: . . . }</pre>	<pre> cmp \$3, %eax ja .Ldef jmp *.Ltab(, %eax, 4) .Ltab: .long .Lcase0 .long .Ldef .long .Lcase2 .long .Lcase3</pre>

Zacznijmy od tego, że nie, teraz to już nie jest ta sama implementacja XD

Po prawej mamy zdefiniowany j = 3 dlatego właśnie taki trafia do akumulatora

Tomczak wspomina też o wypełnianiu flagami rejestru flag czego tu nie widać, może to mieć związek z domyślnym kodem, którego skok bada fali CF i ZF właśnie.

Dalej ja .Ldef - skaczemy do miejsca domyślnego zawierającego fragment kodu od etykiety .Lcase0

Jeśli nie jest to skok domyślny to z pamięci od miejsca .Ltab "będą odczytywane w kolejnych komórkach pamięci adresy docelowych fragmentów kodu zawierające implementacje tych poszczególnych elementów".

Pobieramy albo:

- zerowe przesunięcie względem tej etykiety czyli adres .Lcase0
- Dwójkowe z .Lcase2
- Trójkowe z .Lcase3

Z odpowiedniego miejsca w pamięci zostanie pobrany adres i potraktowany jako docelowy adres kodu.

Gwiazdka oznacza tryb adresowania pośredni bezwzględny.

Dlaczego nie ma gwiazdki przy ja? TURBO WAŻNE!!!!!!!

Etykieta jest maszynowo wstawiana w miejsce, jest to więc skok względny, adresacja bezpośrednia w kodzie.

Natomiast program musi dopiero wyliczyć adres przy jmp. *.Ltab(, %eax, 4). * stosujemy tam gdzie musimy adres wyliczyć, program musi dotrzeć do niego. PRZY TYM DOTYCZY TO TYLKO KOMÓREK W PAMIĘCI!!!!!!! Skojarz sobie z trzema królami, którzy musieli dotrzeć do betlejem. Gwiazda prowadzi, gwiazda oznacza **długą** drogę. Jeśli używamy rejestru należy użyć samego rejestru jako argumentu rozkazu skoku!!!!!! (Swoją drogą droga do rejestrów to jak skoczenie do żabki, a nie podróż trzech króli)

Treść w poprzedniej sekcji tego wykładu.

Jcc—Jump if Condition Is Met

Opcode	Instruction	Description
77 <i>cb</i>	JA <i>rel8</i>	Jump short if above (CF=0 and ZF=0)
73 <i>cb</i>	JAE <i>rel8</i>	Jump short if above or equal (CF=0)
72 <i>cb</i>	JB <i>rel8</i>	Jump short if below (CF=1)
76 <i>cb</i>	JBE <i>rel8</i>	Jump short if below or equal (CF=1 or ZF=1)
72 <i>cb</i>	JC <i>rel8</i>	Jump short if carry (CF=1)
E3 <i>cb</i>	JCXZ <i>rel8</i>	Jump short if CX register is 0
E3 <i>cb</i>	JECXZ <i>rel8</i>	Jump short if ECX register is 0
74 <i>cb</i>	JE <i>rel8</i>	Jump short if equal (ZF=1)
7F <i>cb</i>	JG <i>rel8</i>	Jump short if greater (ZF=0 and SF=OF)
7D <i>cb</i>	JGE <i>rel8</i>	Jump short if greater or equal (SF=OF)
7C <i>cb</i>	JL <i>rel8</i>	Jump short if less (SF<OF)
7E <i>cb</i>	JLE <i>rel8</i>	Jump short if less or equal (ZF=1 or SF<OF)
76 <i>cb</i>	JNA <i>rel8</i>	Jump short if not above (CF=1 or ZF=1)
72 <i>cb</i>	JNAE <i>rel8</i>	Jump short if not above or equal (CF=1)
73 <i>cb</i>	JNB <i>rel8</i>	Jump short if not below (CF=0)
77 <i>cb</i>	JNBE <i>rel8</i>	Jump short if not below or equal (CF=0 and ZF=0)
73 <i>cb</i>	JNC <i>rel8</i>	Jump short if not carry (CF=0)
75 <i>cb</i>	JNE <i>rel8</i>	Jump short if not equal (ZF=0)
7E <i>cb</i>	JNG <i>rel8</i>	Jump short if not greater (ZF=1 or SF<OF)
7C <i>cb</i>	JNGE <i>rel8</i>	Jump short if not greater or equal (SF<OF)
7D <i>cb</i>	JNL <i>rel8</i>	Jump short if not less (SF=OF)
7F <i>cb</i>	JNLE <i>rel8</i>	Jump short if not less or equal (ZF=0 and SF=OF)
71 <i>cb</i>	JNO <i>rel8</i>	Jump short if not overflow (OF=0)
7B <i>cb</i>	JNP <i>rel8</i>	Jump short if not parity (PF=0)
79 <i>cb</i>	JNS <i>rel8</i>	Jump short if not sign (SF=0)
75 <i>cb</i>	JNZ <i>rel8</i>	Jump short if not zero (ZF=0)
70 <i>cb</i>	JO <i>rel8</i>	Jump short if overflow (OF=1)
7A <i>cb</i>	JP <i>rel8</i>	Jump short if parity (PF=1)
7A <i>cb</i>	JPE <i>rel8</i>	Jump short if parity even (PF=1)
7B <i>cb</i>	JPO <i>rel8</i>	Jump short if parity odd (PF=0)
78 <i>cb</i>	JS <i>rel8</i>	Jump short if sign (SF=1)
74 <i>cb</i>	JZ <i>rel8</i>	Jump short if zero (ZF = 1)
0F 87 <i>cw/cd</i>	JA <i>rel16/32</i>	Jump near if above (CF=0 and ZF=0)
0F 83 <i>cw/cd</i>	JAE <i>rel16/32</i>	Jump near if above or equal (CF=0)
0F 82 <i>cw/cd</i>	JB <i>rel16/32</i>	Jump near if below (CF=1)
0F 86 <i>cw/cd</i>	JBE <i>rel16/32</i>	Jump near if below or equal (CF=1 or ZF=1)
0F 82 <i>cw/cd</i>	JC <i>rel16/32</i>	Jump near if carry (CF=1)
0F 84 <i>cw/cd</i>	JE <i>rel16/32</i>	Jump near if equal (ZF=1)

CMP—Compare Two Operands

Opcode	Instruction	Description
3C <i>ib</i>	CMP AL, <i>imm8</i>	Compare <i>imm8</i> with AL
3D <i>iw</i>	CMP AX, <i>imm16</i>	Compare <i>imm16</i> with AX
3D <i>id</i>	CMP EAX, <i>imm32</i>	Compare <i>imm32</i> with EAX
80 /7 <i>ib</i>	CMP <i>r/m8</i> , <i>imm8</i>	Compare <i>imm8</i> with <i>r/m8</i>
81 /7 <i>iw</i>	CMP <i>r/m16</i> , <i>imm16</i>	Compare <i>imm16</i> with <i>r/m16</i>
81 /7 <i>id</i>	CMP <i>r/m32</i> , <i>imm32</i>	Compare <i>imm32</i> with <i>r/m32</i>
83 /7 <i>ib</i>	CMP <i>r/m16</i> , <i>imm8</i>	Compare <i>imm8</i> with <i>r/m16</i>
83 /7 <i>ib</i>	CMP <i>r/m32</i> , <i>imm8</i>	Compare <i>imm8</i> with <i>r/m32</i>
38 /r	CMP <i>r/m8</i> , <i>r8</i>	Compare <i>r8</i> with <i>r/m8</i>
39 /r	CMP <i>r/m16</i> , <i>r16</i>	Compare <i>r16</i> with <i>r/m16</i>
39 /r	CMP <i>r/m32</i> , <i>r32</i>	Compare <i>r32</i> with <i>r/m32</i>
3A /r	CMP <i>r8</i> , <i>r/m8</i>	Compare <i>r/m8</i> with <i>r8</i>
3B /r	CMP <i>r16</i> , <i>r/m16</i>	Compare <i>r/m16</i> with <i>r16</i>
3B /r	CMP <i>r32</i> , <i>r/m32</i>	Compare <i>r/m32</i> with <i>r32</i>

Description

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The CMP instruction is typically used in conjunction with a conditional jump (*Jcc*), condition move (*CMOVcc*), or *SETcc* instruction. The condition codes used by the *Jcc*, *CMOVcc*, and *SETcc* instructions are based on the results of a CMP instruction. Appendix B, *EFLAGS Condition Codes*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, shows the relationship of the status flags and the condition codes.

TEST—Logical Compare

Opcode	Instruction	Description
A8 <i>ib</i>	TEST AL, <i>imm8</i>	AND <i>imm8</i> with AL; set SF, ZF, PF according to result
A9 <i>iw</i>	TEST AX, <i>imm16</i>	AND <i>imm16</i> with AX; set SF, ZF, PF according to result
A9 <i>id</i>	TEST EAX, <i>imm32</i>	AND <i>imm32</i> with EAX; set SF, ZF, PF according to result
F6 /0 <i>ib</i>	TEST <i>r/m8</i> , <i>imm8</i>	AND <i>imm8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result
F7 /0 <i>iw</i>	TEST <i>r/m16</i> , <i>imm16</i>	AND <i>imm16</i> with <i>r/m16</i> ; set SF, ZF, PF according to result
F7 /0 <i>id</i>	TEST <i>r/m32</i> , <i>imm32</i>	AND <i>imm32</i> with <i>r/m32</i> ; set SF, ZF, PF according to result
84 /r	TEST <i>r/m8</i> , <i>r8</i>	AND <i>r8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result
85 /r	TEST <i>r/m16</i> , <i>r16</i>	AND <i>r16</i> with <i>r/m16</i> ; set SF, ZF, PF according to result
85 /r	TEST <i>r/m32</i> , <i>r32</i>	AND <i>r32</i> with <i>r/m32</i> ; set SF, ZF, PF according to result

Description

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

Tomczak mówi o ustawianiu rozkazu CMP lub TEST jako przygotowanie do wykonania właściwego skoku - ustawienie odpowiednich flag, z których skorzystamy. Te rozkazy są niezależne, ale pierwsze ustawiają odpowiednio flagi. Ten zespół jest jak papier kamień nożyce. Jak Wielka Trójca.

Uwaga! Są osobne rozkazy skoków warunkowych kiedy porównywane były argumenty traktowane jak naturalnym, a inne w U2. Bo tu są inne warunki jaki kod dwójkowy odpowiada liczbom większym lub mniejszym.

- W naturalnym binarnym one są po kolei. Używamy "above" i "below" JA, i JB
- W uzupełnieniowym pełnym - "greater" "lesser" JG i JL

Warto zauważyć, że instrukcja CMP ma duży zestaw argumentów pozwalających na porównywanie "czegoś" z argumentami natychmiastowymi

Jak działa porównywanie:

W AT&T Odjęcie pierwszego operandu od drugiego! W intelu na odwrót! + ustawienie flag,

Jeśli argument natychmiastowy jest używany jako operand to jest rozszerzany jak w U2!!!!

Wynik odejmowania odrzucamy!

Instrukcja test wykonuje logiczny AND i ustawia trzy flagi: SF - flagę znaku (jaki jest najbardziej znaczący bit w wyniku tego AND-A), ZF- flagę zera (czy wynikiem tego AND-A jest zero) i PF- flagę parzystości (czy wynikiem AND-A jest słowo zawierające parzystą liczbę jedynek)

LOOP

piątek, 21 czerwca 2024

22:32

LOOP/LOOP_{cc}—Loop According to ECX Counter

Opcode	Instruction	Description
E2 <i>cb</i>	LOOP <i>rel8</i>	Decrement count; jump short if count ≠ 0
E1 <i>cb</i>	LOOPE <i>rel8</i>	Decrement count; jump short if count ≠ 0 and ZF=1
E1 <i>cb</i>	LOOPZ <i>rel8</i>	Decrement count; jump short if count ≠ 0 and ZF=1
E0 <i>cb</i>	LOOPNE <i>rel8</i>	Decrement count; jump short if count ≠ 0 and ZF=0
E0 <i>cb</i>	LOOPNZ <i>rel8</i>	Decrement count; jump short if count ≠ 0 and ZF=0

Description

Performs a loop operation using the ECX or CX register as a counter. Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop. If the address-size attribute is 32 bits, the ECX register is used as the count register; otherwise the CX register is used.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of −128 to +127 are allowed with this instruction.

Some forms of the loop instruction (LOOP_{cc}) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition¹²⁰ code (*cc*) is associated with each instruction to indicate the condition being tested for. Here, the

Rozkaz pętli czyli skoku warunkowego sprawdzającego ecx, dekrementujący wartość, którą przechowuje i dopiero potem sprawdzający czy jest zerem!

UWAGA!!! Argumentem może być tylko przesunięcie i co jeszcze ciekawsze tylko 8 bitowe czyli +127 -128 bajtów!

DODATKOWO możemy jeszcze zrobić zależność od flagi ZF i skończyć pętlę szybciej.

PYTANIE: Czym się różni LOOPE od LOOPNE i LOOPZ od LOOPNZ?

ODPOWIEDŹ: NICZYM.

JEST NADMIAR W MEMONIKACH!

- **zapełnienie:** **for** *i*:= *st* **step** −1 **until** *end* **do** *polecenie* (*i*)

	Intel (MASM)	AT&T/Linux/UNIX	Komentarz
	mov ecx, end	movl \$end, %ecx	
	sub ecx, st-1	subl	
powt:	<i>polecenie (i)</i>	movl zm(%esi,4), %ecx	
		xorl \$mask, %ecx	
		movl %ecx, zm(%esi,4)	
	<i>i:=i-1</i>	decl %esi	
	loop powt	loop powt	

– **zapełnienie**: **for** *i:= st step -1 until end do polecenie (i)*

	Intel (MASM)	AT&T/Linux/UNIX	Komentarz
	mov ecx, end	movl \$end, %ecx	
	sub ecx, st-1	subl	
powt:	<i>polecenie (i)</i>	movl zm(%esi,4), %ecx	
		xorl \$mask, %ecx	
		movl %ecx, zm(%esi,4)	
	<i>i:=i-1</i>	decl %esi	
	loop powt	loop powt	

Jak widać tutaj na tym przykładzie mamy coś w rodzaju pętli for, do ecx na początku wsadzamy wartość końcową, a jako st wartość początkową, od wartości końcowej odejmujemy wartość początkową -1 w ecx i wychodzi nam ilość pętli.

Jako, że LOOP jest skokiem warunkowym i zazwyczaj jest używane na końcu powtarzanego kodu kiedy zostanie skończona zaczynają się wykonywać instrukcje zaraz po niej.

Gdzieś tam jeszcze jest pętla w górę.

IF

piątek, 21 czerwca 2024 22:57

Za pomocą skoków możemy zrobić ifa. W tym celu określamy warunek przeciwny do tego, który chcemy sprawdzać i omijamy dane instrukcje jeśli on wystąpi, bo to oznacza, że nie wystąpił if.

– **akcja warunkowa**: **if** *warunek*=TRUE **then** *polecenie_T* **else** *polecenie_F*

przykład: jeśli (ebx)≤(eax) zmniejsz eax, w przeciwnym razie zwiększ eax

Intel (MASM)	AT&T/Linux/UNIX	Komentarz
cmp arg1, arg2	cmpl %eax, %ebx	# (ebx) ≤ (eax) → ZF=1 ∨ CF=1 (be =T)
jwar alt	jbe alt	#
<i>polecenie F</i>	incl %eax	# (ebx) > (eax), be =F (<i>polecenie_F</i>)
jmp cont	jmp cont	
alt: <i>polecenie T</i>	alt: decl %eax	# (ebx) ≤ (eax), be =T (<i>polecenie_T</i>)
cont:	cont:	

– **ominięcie**: **if** *warunek*=TRUE **then** *polecenie T*

przykład: jeśli (ebx)≤5 wpisz wartość „wart” do ebx

Intel (MASM)	AT&T/Linux/UNIX	Komentarz
cmp arg1, arg2	cmpl \$5, %ebx	# (ebx) ≤ 5 → ZF=1 ∨ CF=1 (gt =F)
jnot_war alt	jgt alt	
<i>polecenie T</i>	movl \$, %ebx	# (ebx) ≤ 5, ngt =T (<i>polecenie_T</i>)
alt:	alt:	# ngt =F

W pierwszym przypadku na obrazku wyżej skaczemy jeśli jmp jest spełniony, przechodzimy do tego co ma się wykonać i lecimy dalej z kodem. Ale dla else musimy ten fragment ominąć, żeby się nie zrobił więc musimy go przeskoczyć.

W drugim przypadku sprawdzamy przeciwny warunek i skaczemy jeśli warunek nie zaszedł czyli z przeciwnego mieliśmy 1

Podsumowując można:

- Sprawdzać warunek i skakać od razu jeśli jest spełniony lub nie skoczyć, tam schowany jest else po czym wyskoczyć z tego elsa tak by nie zrobić ifa.
- Sprawdzać warunek przeciwny i jeśli to będzie jeden skakać wtedy, żeby ominąć tę część kodu ifową

Do samodzielnej analizy!

Jbe - below equal (mniejsze lub równe) w naturalnym binarnym

Cmpl - odejmujemy od drugiego pierwszy

WSPOMNIENIA

piątek, 21 czerwca 2024 23:34

Indeksowanie zmiennej i organizacja pętli (1)

- działanie podstawowe algorytmu

(etykieta)	Rozkaz	Komentarz
	[movl \$wsk, %esi]	# wybrana wartość wskaźnika (do testu)
	movb buf(,%esi,1), %a1	# argument1 (bajt) z pamięci do rejestru
	movb tab(,%esi,1), %b1	# argument2 (bajt) z pamięci do rejestru
	...	# przetwarzanie
	...	# wynik w rejestrze ah
	movb %ah, wyn(,%esi,1)	# wynik (1 bajt) do pamięci

- indeksacja (wspólny indeks)

(etykieta)	Rozkaz	Komentarz
	[inc %esi]	# (aktualizacja wskaźnika) - preindeksacja
	movb buf(,%esi,1), %a1	# argument1 (bajt) z pamięci do rejestru
	movb tab(,%esi,1), %b1	# argument2 (bajt) z pamięci do rejestru
	...	# wynik w rejestrze ah
	movb %ah, wyn(,%esi,1)	# wynik (1 bajt) do pamięci
	[inc %esi]	# (aktualizacja wskaźnika) - postindeksacja

© JANUSZ BIERNAT, AK2-1 -PROGRAMOWANIE'19,

19 MARCA 2019

125

To jest wspomniane.

Indeksowanie zmiennej i organizacja pętli (2)

- utworzenie pętli

(etykieta)	Rozkaz	Komentarz
		#
	movs \$init, %esi	# inicjalizacja wskaźnika
pocz:	...	# stałe parametry jednego przebiegu
	[incl/decl %esi]	# (aktualizacja wskaźnika) - preindeksacja
	movb buf(,%esi,1), %a1	# argument1 (bajt) z pamięci do rejestru
	movb tab(,%esi,1), %b1	# argument2 (bajt) z pamięci do rejestru
	...	# wynik w rejestrze ah
	movb %ah, wyn(,%esi,1)	# wynik (1 bajt) do pamięci
	[incl/decl %esi]	# (aktualizacja wskaźnika) - postindeksacja
	...	# stałe parametry jednego przebiegu
	cmpl \$zakres, %esi	# (esi) - zakres → F (warunek cc)
	jcc pocz	

To jest wspomniane (esi z warunkowym skokiem do stworzenia pętli), tutaj też lea jest, w 3 wykładzie w sekcji INC i LEA jest wyjaśnienie.

Przykład: algorytm Euklidesa ($NWP(a,b)=NWP(b,a \bmod b)$)

(etykieta)	Rozkaz	Komentarz
gcd:	push %ebp	# GCD(a,b)=GCD(b, a mod b)
	movl %esp, %ebp	
	movl 8(%ebp), %eax	# argument „a” w rejestrze eax
	movl 12(%ebp), %ebx	# argument „b” w rejestrze ebx
pocz:	movl \$0, %edx	# 0 do edx
	divl %ebx	# reszta a mod b w edx
	movl %ebx, %eax	# b w miejsce a (do eax)
	movl %edx, %ebx	# a mod b w miejsce b (do ebx)
	andl %edx, %edx	# czy reszta = 0
	jnz pocz	# powtarzaj dopóki reszta ≠ 0
	movl %eax, 8(%ebp)	# GCD(a,b) z rejestru eax na stos
end:	movl %ebp, %esp	
	pop %ebp	
	ret	

rozwiązanie alternatywne to (tylko fragment zaciemniony)

pocz:	xchg %ebx, %eax	# dopóki różnica dodatnia
	subl %ebx, %eax	
	ja rem	# różnica ujemna, korekcja reszty
	addl %ebx, %eax	
	jnz pocz	# powtarzaj dopóki reszta ≠ 0

To jest wspomniane.

TU WSZĘDZIE MOGĄ BYĆ LUKI I BŁĘDY ALE NA TEJ PODSTAWIE TOMCZAKOWI UDAŁO SIĘ SKONSTRUOWAĆ DZIAŁAJĄCE ALGORYTMY

Przykład: wytworzenie liczby przeciwnej do danej dużej liczby (N słów)

inverse:	push %ebp	#
	movl %esp, %ebp	
	movl 8(%ebp), %ebx	# adres liczby (tablicy)
	movl 12(%ebp), %ecx	# rozmiar liczby
	movl \$-1, %esi	# -X = not(X)+ulp
	stc	# ustawienie CF=1 (clc)
pocz:	inc %esi	#
	not (%ebx, %esi,4)	
	adc \$0, (%ebx, %esi,4)	
	jnc koniec	
	loop pocz	# licznik pętli w %ecx
koniec:	movl %ebp, %esp	
	pop %ebp	
	ret	

rozwiązanie alternatywne to (tylko fragment zaciemniony)

	movl \$-1, %esi	# -X = 0-X
	clc	# ustawienie CF=0
pocz:	inc %esi	#
	movl \$0, %eax	
	sbb (%ebx,%esi,4), %eax	# (0-(%ebx,%esi,4) do eax
	movl %eax, (%ebx,%esi,4)	
	loop pocz	# licznik pętli w %ecx

To jest wspomniane.

Przypomiany Big Endian i Little Indian, któryś tu jest wykorzystywany.

CALL (ON ME) & RET

piątek, 21 czerwca 2024 23:54

CALL—Call Procedure

Opcode	Instruction	Description
E8 <i>cw</i>	CALL <i>rel16</i>	Call near, relative, displacement relative to next instruction
E8 <i>cd</i>	CALL <i>rel32</i>	Call near, relative, displacement relative to next instruction
FF /2	CALL <i>r/m16</i>	Call near, absolute indirect, address given in <i>r/m16</i>
FF /2	CALL <i>r/m32</i>	Call near, absolute indirect, address given in <i>r/m32</i>
9A <i>cd</i>	CALL <i>ptr16:16</i>	Call far, absolute, address given in operand
9A <i>cp</i>	CALL <i>ptr16:32</i>	Call far, absolute, address given in operand
FF /3	CALL <i>m16:16</i>	Call far, absolute indirect, address given in <i>m16:16</i>
FF /3	CALL <i>m16:32</i>	Call far, absolute indirect, address given in <i>m16:32</i>

Description

Saves procedure linking information on the stack and branches to the procedure (called procedure) specified with the destination (target) operand. The target operand specifies the address of the first instruction in the called procedure. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of calls:

- Near call—A call to a procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment call.
- Far call—A call to a procedure located in a different segment than the current code segment, sometimes referred to as an intersegment call.
- Inter-privilege-level far call—A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- Task switch—A call to a procedure located in a different task.

132

Instrukcja dostaje jako argument albo przesunięcie względne albo adres bezwzględny pośredni, patrz sekcja: "O skoku jeszcze raz".

Zapisuje na stosie adres instrukcji występującej bezpośrednio po instrukcji call.

CALL

Near Call.

- When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) onto the stack (for use later as a return-instruction pointer).
- The processor then branches to the address in the current code segment specified with the target operand.
- The target operand specifies either an absolute offset in the code segment (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register. which points to the instruction following the CALL

segment, or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register, which points to the instruction following the CALL instruction). The CS register is not changed on near calls.¹³³

RET—Return from Procedure

Opcode	Instruction	Description
C3	RET	Near return to calling procedure
CB	RET	Far return to calling procedure
C2 <i>iw</i>	RET <i>imm16</i>	Near return to calling procedure and pop <i>imm16</i> bytes from stack
CA <i>iw</i>	RET <i>imm16</i>	Far return to calling procedure and pop <i>imm16</i> bytes from stack

Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- Near return—A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- Far return—A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- Inter-privilege-level far return—A far return to a different privilege level than that of the currently executing program or procedure.

134

Powinno być zawsze na końcu w ciele funkcji.

UWAGA! Ze szczytu stosu jest zdejmowany adres powrotu i instrukcja skacze pod ten adres.

RET

- When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer.
- The CS register is unchanged.

Funkcje można wywoływać z dowolnego miejsca innego kodu, powinno się zagwarantować, że funkcja będzie mogła wrócić. Nie zapominajmy, że możemy modyfikować całą pamięć, a więc możemy zepsuć pamięć, w której były instrukcje po CALL. To znaczy "NIE MOŻEMY", ale czaicie o co chodzi. Wszystko na logikę. Byle nie szkodzić.

Akumulator użyty w funkcji będzie tym samym akumulatorem co ten użyty przed jej wywołaniem
inaczej niż podczas deklarowania lokalnych zmiennych w c także uważać!

ABIGAIL I RAMKA STOSU

sobota, 22 czerwca 2024 00:10

Dla przypomnienia:

ABI opisuje sposoby konstruowania kodu i to jak powinien się zachowywać. Programy linuxowe są konstruowane mniej więcej zgodnie z tymi regułami. Stosowanie się do API gwarantuje nam, że kod skompilowany w C na maszynie zgodnej z tym API będzie się tak zachowywał i możemy się z nim dogadać z poziomu assemblera albo napisać kod w assemblerze, który możemy wywołać z poziomu programu w c.

Co definiuje ABI?

a) Typy rodzajów danych w naszym ABI 32-bitowym Linuxa:

ABI

Type	C	sizeof	Alignment (bytes)	Intel386 Architecture
Integral	char	1	1	signed byte
	signed char			
	unsigned char	1	1	unsigned byte
	short	2	2	signed halfword
	signed short			
	unsigned short	2	2	unsigned halfword
Integral	int	4	4	signed word
	signed int			
Integral	long			
	signed long			
Integral	enum			
	unsigned int	4	4	unsigned word
Integral	unsigned long			
Pointer	<i>any-type</i> *	4	4	unsigned word
	<i>any-type</i> (*) ()			
Floating-point	float	4	4	single-precision (IEEE)
	double	8	4	double-precision (IEEE)
	long double	12	4	extended-precision (IEEE)

Jeśli napiszemy kod w c i skompilujemy go na maszynie zgodnej z tym ABI takie będą wartości. Dla innej inne.

b) Jak układać dane w pamięci dla struktur:

Aggregates and Unions

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned component. The size of any object, including aggregates and unions, is always a multiple of the object's alignment. An array uses the same alignment as its elements. Structure and union objects can require padding to meet size and alignment constraints. The contents of any padding is undefined.

- An entire structure or union object is aligned on the same boundary as its most strictly aligned member.
- Each member is assigned to the lowest available offset with the appropriate alignment. This may require *internal padding*, depending on the previous member.
- A structure's size is increased, if necessary, to make it a multiple of the alignment. This may require *tail padding*, depending on the last member.

Figure 3-5: Internal and Tail Padding

```
struct {  
    char   c;  
    double d;  
    short  s;  
};
```

Word aligned, sizeof is 16

pad	1	c	0
d			4
d			8
pad	14	s	12

140

Co możemy wyczytać?

Cała struktura musi być wyrównana do rozmiaru największego z elementów.

UWAGA! W architekturze x86 to są zawsze 4 bajty, wyrównanie nawet 8 i 4-bajtowych słów jest 4-bajtowe

Pad - przesunięcie jednego bajtu.

Co warto zauważyć, a może być nieintuicyjne, że tabelę wyżej uzupełniamy od prawej do lewej w dół. Tak jak rosną adresy.

c) O funkcjach:

Z naszego punktu widzenia funkcje będziemy konstruowali w taki sposób, żeby zarówno do przekazywania argumentów do funkcji jak i gwarantowania, że funkcja nie zepsuje różnych rzeczy związanych z otaczającym ją kodem będziemy używali stosu.

To co będziemy układali na stosie i to w jaki sposób jest to ułożone będzie opisane przez specjalną strukturę - ramkę stosu.

Ramka stosu (*ang. stack frame*)

Figure 3-15: Standard Stack Frame

Position	Contents	Frame	
$4n+8$ (%ebp)	argument word n	Previous	<i>High addresses</i>
...	...		
8 (%ebp)	argument word 0		
4 (%ebp)	return address	Current	
0 (%ebp)	previous %ebp (optional)		
-4 (%ebp)	unspecified		
...	...		
0 (%esp)	variable size		<i>Low addresses</i>

Na szczycie stosu jest argument pierwszy! W argumentcie pierwszym możemy zapisać ile jest i jakie są argumenty i można konstruować funkcje z dowolną liczbą argumentów. Czy to scanf print czy inne, wszystkie funkcje przyjmują jako pierwszy argument ile będzie argumentów i jakie.

1. Ten kto wywołuje funkcję odkłada na stos argumenty tej funkcji.
2. Wykonywana jest instrukcja CALL, która odkłada na stos adres powrotu, w tym ABI wskaźniki są czterobajtowe więc adres też
3. Wnętrze naszej funkcji:
 - a. Na stos odkładana jest wartość rejestru EBP (zauważyć, że jeśli przed wyjściem go zdejmujemy ze stosu to nie zmienimy rejestru EBP czyli odtworzymy jego wartość sprzed wywołania funkcji) Stosuje się sztuczkę, że do ebp po rozkazie RET przepisuje się wskaźnik stosu. Wtedy w EBP mamy adres tego miejsca na stosie. Dzięki temu możemy potem używać stosu jak chcemy, wskaźnik stosu będzie się zmieniał, a w EBP cały czas będziemy mieli adres tego miejsca na stosie. Jeśli chcemy się odwołać do adresu powrotu lub argumentu funkcji (znacznie bardziej częste) to możemy ten adres w prosty sposób wytwarzać jako jakieś przesunięcie względem EBP. Nawet jeśli wskaźnik stosu się zmienia.

Uwaga! Moglibyśmy zrezygnować z EBP i wytwarzać adresy kolejnych elementów względem obecnego wskaźnika stosu. Tylko musimy się nie pomylić. Mamy wtedy jeden dodatkowy rejestr.

Użycie EBP jest wskazane przy debugowaniu kodu. W EBP mamy informację jaki był adres tego miejsca funkcji wywołującej i możemy się po tych wskaźnikach wycofywać w całym łańcuchu wywołań funkcji, debuggerom jest wtedy znacznie łatwiej.

W opcjach kompilatora gcc jest opcja "f omit frame pointer" i "f no omit frame pointer". Tymi opcjami można sobie włączać i wyłączać czy chcemy używać EBP tylko do tego celu i mieć łatwy program do debuggowania czy chcemy je wykorzystać żeby mieć może trochę szybszy kod.

Natomiast z EBP jest dużo łatwiej!

Stos zgodny z ABI musi być wyrównany do szerokości słowa.
Są obszary na stosie, których można używać jak się chce.

Ile mamy dostępnego stosu?

W systemach jest ustawiony limit na rozmiar stosu ok. kilka MB. Takim poleceniem można go

zmniejszyć, zwiększyć użytkownikom się nie da:

```
$ ulimit -a
```

```
real-time non-blocking time  (microseconds, -R) unlimited
core file size               (blocks, -c) 0
data seg size                (kbytes, -d) unlimited
scheduling priority          (-e) 0
file size                    (blocks, -f) unlimited
pending signals              (-i) 253466
max locked memory            (kbytes, -l) 8126828
max memory size              (kbytes, -m) unlimited
open files                   (-n) 1024
pipe size                    (512 bytes, -p) 8
POSIX message queues         (bytes, -q) 819200
real-time priority           (-r) 0
stack size                   (kbytes, -s) 8192
cpu time                     (seconds, -t) unlimited
max user processes           (-u) 253466
virtual memory               (kbytes, -v) unlimited
file locks                   (-x) unlimited
$
```

Jeśli chcielibyśmy zadeklarować lokalną zmienną w funkcji, która by była większa niż 8MB. Tablicę na 10mln elementów to się po prostu nie zmieści. Bo zmienne lokalne funkcji są alokowane na stosie. Na stosie są też adresy powrotu, dla głębokich rekurencji może być to też pamięciożerne.

Wszystkie rejestry w procesorze są jedne! Kiedy wywołujemy funkcję przełączamy się na inny zestaw rejestrów. To się nazywa okna rejestrowe.

Trzeba ustalić, które rejestry funkcja może zmieniać, a których nie może.

Konwencja wywoływania funkcji

- All registers on the Intel386 are global and thus visible to both a **calling** and a **called** function.
- Registers `%ebp`, `%ebx`, `%edi`, `%esi`, and `%esp` “belong” to the **calling** function. In other words, a called function must **preserve** these registers’ values for its caller.
- **Remaining** registers “belong” to the **called** function. If a calling function wants to preserve such a register value across a function call, it must save the value in its local stack frame.

144

WYKUĆ NA BLACHĘ!

EBP, EBX, EDI, ESI, ESP należą do funkcji wywołującej, tej zewnętrznej, która wywołuje. Funkcja wywoływana musi zagwarantować, że po wyjściu z tej funkcji te rejestry będą miały taką samą wartość jak przed wejściem do tej funkcji. To nie znaczy, że nie można w niej tych rejestrów używać. Najczęściej zapisuje się wartości tych rejestrów na stosie. I odtwarza przed wyjściem z funkcji. Jednak nie wszystkich, nie esp, to jest wskaźnik stosu, jak on będzie źle ustawiony to nie zadziała instrukcja RET. Instrukcja RET zdejmuję ze stosu adres powrotu. Gdy w momencie wychodzenia z funkcji wskaźnik stosu nie będzie wskazywał poprawnie instrukcja RET spróbuje skoczyć w inne miejsce. Wskaźnik stosu musi być poprawny.

Pozostałe funkcje mogą się zmienić.

Jak zatem zwracać wartości z funkcji? Różnie w zależności od tego co zwracamy, jeśli zwracamy całkowitoliczbowe wartości lub wskaźniki to te wartości zwraca akumulator. 64 bitowe wartości są zwracane przez akumulator rozszerzony edx:eax.

Zmiennoprzecinkowe wyniki funkcji są zwracane przez specjalny zestaw rejestrów

Funkcje nie zwracające wartości nie zwracają wartości i tyle.
Funkcje zwracające bardziej zaawansowane struktury muszą mieć miejsce przygotowane przez tego kto te funkcje wywołuje.

Funkcja zgodna z ABI zazwyczaj zaczyna się tak:

prologue:

```
pushl %ebp      / save frame pointer
movl  %esp, %ebp / set new frame pointer
subl  $80, %esp  / allocate stack space
pushl %edi      / save local register
pushl %esi      / save local register
pushl %ebx      / save local register
```

1. `pushl %ebp` - odkładanie na stos rejestru `ebp`
2. `movl %esp, %ebp` - do rejestru `ebp` wkładam wskaźnik stosu (dzięki temu w `ebp` mamy adres tego odłożonego poprzedniego `ebp`)
3. `subl $80, %esp` -alokacja miejsca, przesunięcie wskaźnika stosu w dół (bo stos rośnie w stronę niższych adresów), to jest miejsce na zmienne lokalne funkcji, dzięki temu mamy możliwość rekurencji
4. Dodanie wartości rejestrów, żeby można było je wykorzystywać.

Wewnątrz funkcji można alokować tablice statyczne o rozmiarze dynamicznym zależnym od argumentu do tej funkcji, bo miejsce jest alokowane dopiero podczas rozpoczynania wykonywania funkcji.

1:20:00 Niezrozumiały kod, który Tomczak pisze na tablicy jako dowód, żeby sobie sprawdzić.

I kończy tak:

```
movl %edi, %eax / set up return value
epilogue:
popl %ebx      / restore local register
popl %esi      / restore local register
popl %edi      / restore local register
leave          / restore frame pointer
ret            / pop return address
```

Odtwarzamy wartości tych trzech rejestrów, zwrócić uwagę na kolejność `leave` - to to samo co:

```
{
Movl %ebp %esp - z ebp zwracamy do esp adres odłożonego ebp
Popl %ebp - zdejmujemy ze stosu ebp
}
```

^
|
|
|

TO WSZYSTKO W ZASADZIE TRZEBA ZNAĆ NA BLACHĘ!

ZAPAMIĘTAĆ:

Ebp pełni rolę strażnika pierwszego elementu na stosie czyli adresu zaraz po adresie powrotu! To dzięki niemu możemy się przenieść w to miejsce i skorzystać z RET.

Inty i wskaźniki przekazywane na stos:

Figure 3-21: Integral and Pointer Arguments

Call	Argument	Stack address
g(1, 2, 3, (void *)0);	1	8 (%ebp)
	2	12 (%ebp)
	3	16 (%ebp)
	(void *)0	20 (%ebp)

Jeśli nasza funkcja przyjmuje trzy argumenty całkowitoliczbowe 1,2,3 i jeden wskaźnik (void *) 0 to na stosie argumenty są ułożone w taki sposób jak widać adresy rosną w dół, a na poprzednim obrazku (Figure 3-15) malały, bo stos rośnie w stronę adresów niższych, te adresy są adresami kolejnych coraz wyższych komórek pamięci, które są coraz bliżej dna stosu. (Żeby przyłączyć z poprzednim trzeba odwrócić tabelę (Polska na czele)). Na szczycie stosu jest więc 1.

Ciekawostka (floaty przekazywane na stos):

Figure 3-22: Floating-Point Arguments

Call	Argument	Stack address
h(1.414, 1, 2.998e10);	word 0, 1.414	8 (%ebp)
	word 1, 1.414	12 (%ebp)
	1	16 (%ebp)
	word 0, 2.998e10	20 (%ebp)
	word 1, 2.998e10	24 (%ebp)

Argumenty zmiennoprzecinkowe przekazuje się przez stos, ale wyniki zwraca się przez stos zmiennoprzecinkowy. Pierwszy double zajmuje 8 bajtów, int zajmuje 4 bajty i potem drugi double 8 bajtów.

Struktury przekazywane na stos, obraz pamięci jest odkładany na stos:

Figure 3-23: Structure and Union Arguments

Call	Argument	Callee
i(1, s);	1	8 (%ebp)
	word 0, s	12 (%ebp)
	word 1, s	16 (%ebp)

Rada Tomczaka: Nie przekazujcie skomplikowanych obiektów czy struktur do funkcji. To wolno działa. Nie twórzcie niepotrzebnie kopii waszych obiektów. Może się nawet nie zmieścić Jak się da to referencja.

NA EGZAMIN

sobota, 22 czerwca 2024 00:25

Umieć zaprogramować Super Mario Bros na tym:



Generacje języków programowania

- 1GL - kod maszynowy
- 2GL - język assemblera
 - automatyczne obliczanie adresów (symbole, etykiety), udogodnienia zwiększające czytelność (mnemoniki, komentarze, dyrektywy, makra, literały liczbowe i znakowe) i zarządzanie kodem (łatwe łączenie modułów)
- 3GL
 - C, C++, Java, Python, PHP, Perl, C#, BASIC, Pascal, Fortran, ALGOL, COBOL
 - wysokopoziomowe abstrakcje (programowanie strukturalne i obiektowe) ale dopasowane do ograniczeń maszyny, czytelność, przenośność
 - konieczna translacja (kompilator/interpreter)
- 4GL
 - ABAP, Unix Shell, SQL, PL/SQL, Oracle Reports, R
 - abstrakcje dopasowane do dziedziny (problemów), a nie maszyny
- 5GL
 - Prolog, Clipper, LabView
 - Opis problemu i wymagań nałożonych na wynik, brak algorytmu

150

Dla każdego języka jest przewidziany standard. Jeśli ktoś chce się doskonalić w jakimś powinien znaleźć najnowszą wersję predraftu czyli czegoś co już poszło do zatwierdzenia i jest za darmo.

Ten standard będzie w pewnym sensie niezgodny z ABI.

W praktyce można napisać kod, który będzie się zachowywał jakoś na jakiejś maszynie i tylko tyle. Jak użyjemy tego na innej maszynie to może się zachować inaczej jeśli nie będzie zgodny ze standardem języka, a nie ze standardem ABI. ABI jest zgodne ze standardem języka natomiast nie jest w drugą stronę.

DZIĘKUJĘ BARDZO!