Kryptographie

Jonas Schwambi Schwammberger

4. Semester (FS 2013)

Inhaltsverzeichnis

1 Register

- %esp = stack pointer
- %ebp = base pointer
- %eax = accumulator, return Werte von Funktionen werden hier abgelegt.
- %ebx = base index (array manipulation)
- %ecx = counter (array manipulation)
- %edx = data / general register
- %esi = source index (string manipulation)
- %edi = destination index (string manipulation)
- %eip = instruction pointer

Ausser %eip und %esp sind alles General Purpose Register, man kann auch %ebx für eine Array-Manipulation verwenden.

1.1 MOV Instruktion

movl kann in drei Varianten verwendet werden:

- movl "register", "register"
- movl "register, [Expression]
- movl [Expression], "register"

1.2 Expression

Generelle Funktion für Expressions: $D(Rb, Ri, S) = Mem[Reg[Rb] + S \cdot Reg[Ri] + D]$

- D: Konstante in Byte(4 Byte für 64b)
- Rb: Base Register
- Ri: Index Register, können alle sein ausser %esp und %ebp
- S: Skalar in Zweierpotenz

Beispiele:

Ausdruck	Berechnung	Adresse im Hauptspeicher
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx,%ecx,4)	0xf000 + 4*0x100	0xf400
0x80(,%edx,2)	2*0xf000 + 0x80	0x1e080

2 Function Call

2.1 Stack Frame

%ebp zeigt immer auf die "Basis"des stacks, heisst alle Adressen kleiner als %ebp gehören zur momentan ausgeführten Methode. Die Parameter dieser Methode sind dabei auf den Adressen grösser als %ebp abgespeichert. Die Speicherstelle, auf die %ebp hinzeigt, ist der &ebp Wert der vorherigen Methode. 4(%ebp) beinhaltet die Return-Adresse für diese Methode, alles höher als 4(%ebp) sind Parameter der momentanen Methode.

⋮
yp
xp
ret addr
%ebp ← pushl %ebp %ebp
%ebx

2.2 Function Call Setup

Nachdem der Aufrufer die Parameter auf den Stack abgelegt und "Call Functionäusgeführt hat.

```
pushl %ebp
movl %esp, %ebp
```

2.3 Function Call Teardown

Falls die Methode einen Rückgabewert hat, muss dieser vorher noch in%eax abgelegt werden.

```
movl %ebp, %esp
pop %ebp
return
```

3 Instruktionen

3.1 Arithmetische Operatoren

3.2 Instruktionen für den Methodenaufruf

push Src	
pop Dest	
call (label)	
ret	

3.2.1 Binäre Operatoren

Alle binären Operatoren lesen aus dem Source Register und den berechneten Wert in das Destination Register.

Befehl	Beschreibung
addl	Dest += Source
subl	Dest -= Source
imull	Dest *= Source
sall	Dest << Source
sarl	Dest >> Source, füllt mit 1 auf falls MSB = 1
shrl	Dest >> Source, füllt immer mit 0 auf
leal	siehe LEA Instruction.
xorl	...
andl	...
orl	...

3.2.2 Unäre Operatoren

Befehl	Beschreibung
incl	increment
decl	decrement
negl	negate
notl	not operator

3.2.3 LEA Instruction

Vom Internet: LEA, the only instruction that performs memory addressing calculations but doesn't actually address memory. LEA accepts a standard memory addressing operand, but does nothing more than store the calculated memory offset in the specified register, which may be any general purpose register.
What does that give us? Two things that ADD doesn't provide:
the ability to perform addition with either two or three operands, and the ability to store the result in any register; not just one of the source operands.

4 Vergleiche und Konditionen

Alle Compare Operationen werden durchgeführt, indem verschiedene Flags überprüft werden. Diese Flags werden von den arithmetischen Operationen selber gesetzt, oder durch die Befehle testl oder cmpl. Zum Beispiel überprüft die JUMP ZERO Instruktion, ob das ZERO FLAG von einer anderen Instruktion zuvor gesetzt wurde.

4.1 Flags

Abkürzung	Name	wird gesetzt durch
ZF	Zero Flag	wird von testl gesetzt.
SF	Signed Flag	wird von testl gesetzt.
OF	Overflow Flag	von arithmetischen Operationen gesetzt.
CF	Carry Flag	von arithmetischen Operationen gesetzt.

4.2 Vergleichsoperatoren

cmpl Var1, Var2	Rechnet Var2 - Var1, ohne das Resultat in Var2 zu speichern. Nur die Flags werden verändert. Heisst Var1 == Var2, dann ist das Zero Flag gesetzt.
testl	Macht das gleiche wie cmpl, mit dem Unterschied dass es Bitwise AND anstatt einer Substraktion macht.
cmovle	move src to dest if condition c is true(less or equal).

SetX Befehle verändern die Flags direkt, falls man das möchte:

Befehl	Ausdruck	Beschreibung
sete	ZF	Equal / Zero
setne	$\sim ZF$	Not Equal / Not Zero
sets	SF	Negative
setns	$\sim SF$	Nonnegative
setg	$\sim (SF \hat{} OF) \& \sim ZF$	Greater (Signed)
setge	$\sim (SF \hat{} OF)$	Greater or Equal (Signed)
setl	$(SF \hat{} OF)$	Less (Signed)
setle	$(SF \hat{} OF) \mid ZF$	Less or Equal (Signed)
seta	$\sim CF \& \sim ZF$	Above (unsigned)
setb	CF	Below (unsigned)

4.3 Jump

Befehl	Flags	Beschreibung
jmp (label)	1	Bedingungsloser jump
je (label)	ZF	jump equal or zero
jne (label)	ZF	jump not equal or not Zero
js (label)	SF	jump negative
jns (label)	SF	jump not negative
jg (label)	(SF$\hat{}$OF)& ZF	jump greater
jge (label)	(SF$\hat{}$OF)	jump greater or equal
jl (label)	(SF$\hat{}$OF)	jump less
jle (label)	(SF$\hat{}$F)\ZF	jump less or equal
ja (label)	$\tilde{\text{C}}$F & $\tilde{\text{Z}}$F	jump above (unsigned)
jb (label)	CF	jump below (unsigned)

5 Loops und If's

5.1 If Statement

5.1.1 Unter 32Bit

C Code:

```
int absdiff(int x,int y)
{
        int result;

        if(x > y)
                result = x-y;
```

```
        else
                result y-x;

        return result;
}
```

Assembler:

```
absdiff:
        pushl %ebp
        movl %esp,%ebp

        movl 8(%ebp),%edx
        movl 12(%ebp),%eax
        cmpl %eax,%edx
        jle .L7
        movl %edx,%eax
.L8:
        movl %ebp,%esp
        popl %ebp
        ret
.L7:
        subl %edx, %eax
        jmp .L8
```

5.1.2 Unter 64Bit

C Code, der Selbe wie unter 32 Bit. Assembler:

```
absdiff:
        pushl %ebp
        movl %esp,%ebp

        movl %edi, %eax # v = x
        movl %esi, %edx # ve = y
        subl %esi, %eax # v -= y
        subl %edi, %edx # ve -= x
        cmpl %esi, %edi # x:y
        cmovle %edx %eax # v=ve if <=
        movl %ebp,%esp
        popl %ebp
        ret
```

5.2 Loops

5.2.1 Do While Loops

C Code:

```
int fact(int x)
{
        int result = 1;
```

```
        do
        {
                result *= x;
                x = x-1;
        } while(x > 1);

        return result;
}
```

Intermediate Code, bevor der Code zu Assembler übersetzt wird:

```
int fact(int x)
{
        int result = 1;

        loop:
                result *= x;
                x = x-1;
                if (x > 1)
                        goto loop;

        return result;
}
```

Assembler:

```
fact:
        pushl %ebp
        movl %espm%ebp
        movl $1,%eax
        movl 8(%ebp),%edx
L11:
        imull %edx,%eax
        decl %edx                   # Compare x : 1
        cmpl $1,%edx    # if > goto loop
        jg L11

        movl %ebp,%esp
        popl %ebp
        ret
```

5.2.2 while loops

While loops werden vom GCC in einen Do While loop übersetzt.

Alte Übersetzungsart Pseudocode While:

```
while(TEST)
        Body
```

Pseudo intermediate Code:

```
if(TEST)
        goto DONE
LOOP:
        Body
        if(TEST)
                goto LOOP;
DONE:
```

Neue Übersetzungsart In der neuen Übersetzungsart wird der unnötige Test vor dem eigentlichen Loop weggelassen, heutige Prozessoren haben keine Performance einbussen bei unconditional jumps
Pseudocode:

```
goto MIDDLE
LOOP:
        Body
MIDDLE:
        if(TEST)
                goto LOOP
```

5.2.3 For Loops

For loops sind eigentlich nur While loops mit einer speziellen letzten Zeile. For loops werden in einen While loop umgewandelt, der wieder in einen do while ...
Pseudocode for:

```
for(INIT;TEST;UPDATE)
        body
```

Pseudocode in while übersetzt:

```
INIT
while(TEST)
{
        body
        UPDATE
}
```

5.3 Select Case

Ein Select Case gibt es zwei Möglichkeiten, entweder es wird als eine Reihe von if then else Anweisungen implementiert, was bei vielen Cases sehr langsam wird, oder mittels einer Jump Table. Der GCC entscheidet selbst, was er macht.
Beispiel C Code:

```
typedef enum {ADD,MULT,MINUS,DIV,MOD,BAD} op_type;
char unparse_symbol(op_type op) {
        switch(op)
        {
                case ADD:
                        return '+';
```

```
                case MULT:
                        return '*';
                case MINUS:
                        return '-';
                case DIV:
                        return '/';
                case MOD:
                        return '%';
                case BAD:
                        return '?';
        }
}
```

Jump Table

```
.section .rodata
        .align 4
.L57:
        .long .51       //Addresse fuer Case 0
        .long .L52      //Addresse fuer Case 1
        ...
        .long .L56  //Addresse fuer Case 5
```

Eigentlicher Switch wird ungefähr so übersetzt. Jeder Case bekommt ein eigenes Label. Ebenso wird hier ein weiteres Label hinzugefügt, mitdem wir zu den Instruktionen springen können, die nach dem Switch ausgeführt werden sollen.

```
.L51
        movl $43,\%eax  //'+'
        jmp .L49
.L52
        movl $42,\%eax  //'*'
        jmp .L49
.L53
        movl $45,\%eax  //'-'
        jmp .L49
...

.L49
        movl %ebp, %esp
        popl %ebp
        ret
```

Methodenaufruf, wie die Jump Tabelle und die

```
unparse_symbol:
        pushl %ebp
        movl %esp,%ebp

        movl 8(%ebp), %eax  #eax = op
        cmpl $5,%eax
        ja      .L49                    # if > goto end
        jmp *.L57(,%eax,4)      #  .L57[%eax]
```

Erklärung der Letzten Instruktion: $(,\%eax,4)$ wird übersetzt in $(0 + \%eax) \cdot 4$, somit haben wir unseren Offset für die Jump Tabelle. Der Rest des Ausdrucks bedeutet: Ġehe zur Memory Adresse, die das Label L57 hat, addiere den Offset dazu und springe dan zum Wert, der diese Adresse beinhaltet."Dieser Wert ist dann zum Beispiel die Adresse des Labels .L51.

6 Bitwise Magix

```
int bitXor(int x, int y) {
        return ~x & y;
}

int isEqual(int x, int y) {
        return !(x ^ y);
}
```