CSO: Recitation 10

Goktug Saatcioglu

10.31.2019

1 Continuing Assembly

2 Control

2.1 Processor State

Information about currently executing program

- Temporary data (`%rax`, ...)
- Location of runtime stack (`%rsp`, `%rbp`)
- Location of current code control point (`%rip`)
- Status of recent tests (CF,ZF,SF,OF)

2.2 Condition Codes Can Be Set Implicitly For Arithmetic

Consider the instruction `addq Src, Dest` (`t = a+b`), then for the arithmetic operation the following may be set

- CF (Carry flag) set if carry out from most significant (31-st) bit (unsigned overflow)
- ZF (Zero flag) set if `t == 0`
- SF (Sign flag) set if `t < 0` (as signed)
- OF (Overflow flag) set if signed overflow `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`
- Condition codes not set by lea instruction!

2.3 Logical Operations

For logical operations

- The carry and overflow flags are set to zero.
- For shift instructions:
  - The carry flag is set to the value of the last bit shifted out.
  - Overflow flag is set to zero.

2.4 INC and DEC instructions

- Affect the overflow and zero flags
- Leave carry flag unchanged

2.5 Setting Condition Codes Explicitly

- `cmpl b, a` sets condition codes based on computing `a-b` without storing the result in any destination
  - CF set if carry out from most significant bit (used for unsigned comparisons)
  - ZF set if `a == b`
  - SF set if `(a-b) < 0` (as signed)
  - OF set if `(a-b)` results in signed overflow
- `testq b, a` sets condition codes based on value of `(a & b)` without storing the result in any destination
  - ZF set if `(a & b) == 0`
  - SF set if `(a & b) < 0`

2.6 Important Note

The processor does not know if you are using signed or unsigned integers. OF and CF are set for every arithmetic operation!

2.7 Reading Condition Codes

`setX dest` sets the lower byte of `dest` based on combinations of condition codes and does not alter remaining 7 bytes. Destination can also be memory location. See lecture notes for details on this instruction.

3 Jumps

3.1 Jumping

We can "jump" to different part of code depending on condition codes. This can be used to implement conditional branching, switch statements, for loops and function calls (control flow operations). `jX` where `X` can be many different conditions. See lecture notes.

3.2 Indirect jump

We can also perform indirect jumps. This is given by the isntruction `jmp * Operand` where `jmp` is an unconditional jump and `* Operand` can a register or memory address using any of the addressing modes we saw.

3.3 Bad Cases For Conditional Moves

- Expensive Computations
  - `val = Test(x) ? Hard1(x) : Hard2(x);`
  - Both values get computed
  - Only makes sense when computations are very simple
- Risky Computations
  - val = p ? *p : 0;
  - Both values get computed
  - May have undesirable effects
- Computations with side effects
  - val = x ¿ 0 ? x *= 7 : x += 3;
  - Both values get computed
  - Must be side-effect free

4 Loops

4.1 General "Do-While" Translation

If we have C code of the form

```
  do
    Body
    while (Test);
```

we can translate it into the following goto version

```
loop:
  Body
  if (Test)
    goto loop
```

which then lets us generate assembly for the loop.

4.2 General "While" Translation #1

If we have C code of the form

```
  while (Test) Body
```

we can translate it into the following goto version

```
  goto test;
loop:
  Body
test:
  if (Test)
      goto loop;
done:
```

which then lets us generate assembly for the loop. This translation is called "Jump-to-middle" translation.

4.3 General "While" Translation #2

If we have C code of the form

```
  while (Test) Body
```

we can translate it into the following do-while version

```
  if (!Test)
    goto done;
  do
    Body
  while(Test);
done:
```

which we can translate it into the following goto version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

which then lets us generate assembly for the loop. This translation is called "Do-while" translation.

4.4 For Loops

We saw this in an earlier class, first translate into a while loop and then use one of the above strategies.

4.5 Switch Statement

For switch statements we create what is called a jump table. A jump table is a table of addresses where each entry in the table corresponds to a target address which has code in it. So each case of a switch statement is an entry in jump table and each corresponding body of a case is that entries target. Recall the following:

- Table Structure
  - Each target requires 8 bytes
  - Base address at some label (e.g. `.L4`)
- Jumping
  - Direct: `jmp .L8`
  - Jump target is denoted by label `.L8`
  - Indirect: `jmp *.L4(,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of `8` (addresses are 8 bytes)
  - Fetch target from effective Address `.L4 + x*8`

5 While Loop Exercise

Consider the following C code.

```
int fibn(int n)
{
  int a = 0, b = 1, c;
  if (n == 0)
    return a;
  for (int i = 2; i <= n; i++)
  {
     c = a + b;
     a = b;
     b = c;
  }
  return b;
}
```

As it can be seen, it calculates the n-th Fibonnaci number. We wish to translate this code into asssembly. How should we go about this? Let's begin by converting the for loop into a while loop. We know how to do this from previous classes. What is the translated C code? The answer is given below.

```
int fibn(int n)
{
  int a = 0, b = 1, c;
  if (n == 0)
    return a;
  int i = 2;
  while (i <= n)
  {
     c = a + b;
```

```
    a = b;
    b = c;
    i++;
  }
  return b;
}
```

Next, we should translate this into something that uses goto instead. This is because we want the code to resemble a low-level representation as much as possible so that the translation is easy to do. Can you figure this out? The answer is given below.

```
int fibn(int n)
{
  int a = 0, b = 1, c;
  if (n == 0)
    return a;
  int i = 2;
  goto test;
  loop:
    c = a + b;
    a = b;
    b = c;
    i++;
  test:
    if (i <= n)
      goto loop;
  return b;
}
```

Notice that we picked the first translation strategy to translate the while loop. Now we need to start generating the assembly code. We need to make some assumptions about where the variables are located in memory and which registers we will be using. Since the code is simple enough and we lack an outer context, we will only use registers. Let `n` be stored in `%rdi` and let's use `%rsi` for `a`, `%rax` for `b`, `%r8` for `c` and `%r9` for `i`. Why did we choose to assign `b` to `%rax`? `%rax` is known as the return register and we are more than likely to return `b` so it is a minor optimization. Now try writing the assembly code for this loop. Once you have an implementation ready, check out the solution given below.

```
fib:
  movq $0, %rsi             # a = 0
  movq $1, %rax             # b = 1
  cmpq $0, %rdi             # n == 0
  je   .L1
  movq $2, %r9              # i = 2
  jmp  .L2                  # goto test
.L3:                        # loop
  leaq (%rsi,%rax), %r8     # c = a + b
  movq %rax, %rsi           # a = b
  movq %r8, %rax            # b = c
  inc  %r9                  # i++
.L4:                        # test
  cmpq %rdi, %r9            # cmp n, i
  jle  .L3                  # goto loop
  ret
.L2                         # n == 0
```

```
  movq %rdi, %rax           # rax = a
  ret
```

The only thing to watch out for is the base case. Since assembly code labels are only for jump purposes, the program will continue executing after jumping to a label unless we change the control flow. In this case, we put the base case return code at the very end so if the base case is true we simply jump there, move the result into `%rax` and return. The rest of the code is pretty intuitive and pretty efficient. Compare your solution against this one. What are the similarities and the differences? What might you have gotten wrong? What do you think you might have done better and why is it better?

6 Switch Statement Exercise

Consider the following C code.

```
long switch_eg(long x, long y, long z)
{
  long res = 1;
  switch(x) {
    case 1:
      res = y*z;
      break;
    case 2:
      res = y/z;
      /* Fall Through */
    case 3:
      res += z;
      break;
    case 5:
      /* Fall Through */
    case 6:
      res -= z;
      break;
    default:
      res = 2;
    }
  return res;
}
```

We wish to translate this code into assembly. Let's begin by identifying the `merge` points of this code. These occur when there is a fallthrough case and we need two different initialization codes depending on whether we fell through another or we're just entering some case. For case 3 we see that we need to have `res` initialized to 1. For case 2 this is not necessary. This also occurs for case 5 and case 6 but there are no merge points since case 5 directly falls through into case 6. So we re-write the above code to move initialization code when it is absolutely necessary and use goto to a `merge` label. This is given below.

```
long switch_eg(long x, long y, long z)
{
  long res;
  switch(x) {
    case 1:
      res = y*z;
      break;
    case 2:
      res = y/z;
      goto merge;
```

```
      /* Fall Through */
    case 3:
      res = 1;
      merge:
        res += z;
      break;
    case 5:
      /* Fall Through */
    case 6:
      res = 1;
      res -= z;
      break;
    default:
      res = 2;
    }
  return res;
}
```

Now we can translate this code into a jump table and some assembly. Let's assume `x` is in `%rdi`, `y` is in `%rsi`, `z` is in `%rdx` and `res` is in `%rax`. Then our jump table looks like as follows.

```
.section .rodata
  .align 8
.L4:
  .quad .L8 #default
  .quad .L3 #x=1
  .quad .L5 #x=2
  .quad .L9 #x=3
  .quad .L8 #x=4
  .quad .L7 #x=5
  .quad .L7 #x=6
```

Note how we have 7 different possible values for the cases of the switch statement. 1 through 6 are all contigious numbers so we can lay out all the cases as 8 byte addresses. Anything else should take the default branch so we put that first in the table. In terms of the table, if we for example have case 4 then we would calculate the address with respect to `.L4` and put the address for case 4 4-quad labels below `.L4`. So we can use an indirect jump instruction and calculate the address we need to go to depending on the value of x as `*.L4(,%rdi,8)`. Recall that `%rdi` holds `x`. Translating the rest of the table is as easy as creating all the labels and filling in the code that should go in them. Don't forget the `merge` label we created above. Try translating this code into assembly and then compare against the answer given below.

```
switch_eg:
  movq %rdx, %rcx
  cmpq $6, %rdi       # x:6
  jg .L8              # jump to default
  jmp *.L4(,%rdi,8)
.L3:                  # case 1
  movq %rsi, %rax     # w = y
  imulq %rdx, %rax    # w = w*z
  ret
.L5:                  # case 2
  movq %rsi, %rax     # w = y
  cqto
  idivq %rcx
```

```
  jmp .L6             # goto merge
.L9:
  movl $1, %eax       #  w = 1
.L6:                  # merge:
  addq %rcx, %rax     # w += z
  ret
.L7:                  # case 5,6
  movq $1, %rax       # w = 1
  subq %rdx, %rax     # w -= z
  ret
.L8:                  # default:
  movl $2, %eax       # 2
  ret
```

Notice how the compiler may generate different code sequences to implement more complex control. The above is one possible solution for this switch statement and you could have had many different solutions. How should this code be modified if there were more instructions after the switch statement? Hint: `jmp .L10`.