CS 4400
Computer Systems

LECTURE 6

Representing control flow

The `gdb` *debugger*

Lab 2

- Read *entire* lab2_specs document before starting.
- DO NOT request > 1 bomb. Costly deduction otherwise.
  - If bomb does not download in 10 mins, email teach-cs4400.
- If scoreboard has not updated in 10 mins, email teach-cs4400.
- Avoid accidental explosions!
  - Points lost will not be returned, no matter the reason.
  - If you are new to the tools recommended (like gdb), first experiment with them on code other than your bomb.

Control Flow

- Default for C and assembly code programs is to have control flow sequentially.
  - statements/instructions executed in the order they appear
- In C, conditionals, loops and switches allow control to flow in non-sequential order.
  - exact sequence depends on values of program data
- In assembly code, low-level mechanisms implement non-sequential control flow.
  - jump to a different part of program (may be depend on a test)

Condition Code Registers

- Single-bit condition code registers describe the attributes of the most recent arithmetic or logical operation.
  - can be tested to perform conditional branches
  - CF (carry flag) most recent op generates carry out of MSB
  - ZF (zero flag) most recent op yielded zero
  - SF (sign flag) most recent op yielded a negative value
  - OF (overflow flag) most recent op caused 2's complement OF
- Suppose we used `addl` to perform $t = a + b$.
  - CF: `(unsigned t) < (unsigned a)`
  - ZF: `t == 0` • SF: `t < 0`
  - OF: `(a < 0 == b < 0) && (t < 0 != a < 0)`

Condition Codes

- All integer arithmetic operations (covered in Lec 5) cause the condition codes to be set.
  - `leal` is the exception
- Two more instructions set the condition codes without altering any other registers.
- `cmpl src2,src1` sets the condition codes according to the difference in `src1` and `src2`.
- `testl src2,src1` sets the condition codes according to the AND of `src1` and `src2`.

Accessing Condition Codes

- Rather than access condition codes directly, either they are set to an integer register or a conditional branch is performed based on some combination of the codes.
- Set a single byte to 0 or 1 depending on some combination of condition codes.
  - destination is either single-byte register or memory location
  - to generate 32-bit result, must clear high-order 24 bits
- *Example*:

```
compl %eax,%edx    ;compare b,a
setl %al           ;set< low bits of %eax
movzbl %al,%eax    ;zero remaining bits
```

The `set` Instructions

- `sete` *dst*, “set when equal”, *dst* = ZF
- `setne` *dst*, “set when not equal”, *dst* = ~ZF
- `sets` *dst*, “set when signed”, *dst* = SF
- `setns` *dst*, “set when not signed”, *dst* = ~SF
- `setg` *dst*, “set when greater”, *dst* = ~(SF ^ OF) & ~ZF
- `setge` *dst*, “set when greater or equal”, *dst* = ~(SF ^ OF)
- `setl` *dst*, “set when less”, *dst* = SF ^ OF
- `setle` *dst*, “set when less or equal”, *dst* = (SF ^ OF) | ZF
- `seta` *dst*, “set when above” (unsigned >), *dst* = ~CF & ~ZF
- `setae` *dst*, “set when above or equal” (unsigned ≥), *dst* = ~CF
- `setb` *dst*, “set when below” (unsigned <), *dst* = CF
- `setbe` *dst*, “set when below or equal” (unsigned ≤), *dst* = CF | ZF

Exercise: Comparisons

```
char ctest(int a, int b, int c) {
  char t1 = a __ b;
  char t2 = b __ (    )a;
  char t3 = (    ) c __ (    ) a;
  char t4 = (    ) a __ (    ) c;
  char t5 = c __ b;
  char t6 = a __ 0;
  return t1+t2+t3+t4+t5+t6;
}
```

```
movl 8(%ebp),%ecx        ;get a
movl 12(%ebp),%esi       ;get b
cmpl %esi,%ecx    ;compare a-b
setl %al                    ;t1
cmpl %ecx,%esi    ;compare b-a
setb -1(%ebp)               ;t2
cmpw %cx,16(%ebp);compare c-a
setge -2(%ebp)              ;t3
movb %cl,%dl
cmpb 16(%ebp),%dl;compare a-c
setne %bl                   ;t4
cmpl %esi,16(%ebp)  ;comp c-b
setg -3(%ebp)               ;t5
testl %ecx,%ecx     ;test a&a
setg %dl                    ;t6
addb -1(%ebp),%al       ;t1+=t2
addb -2(%ebp),%al       ;t1+=t3
addb %bl,%al            ;t1+=t4
addb -3(%ebp),%al       ;t1+=t5
addb %dl,%al            ;t1+=t6
movsbl %al,%eax ;convert type
```

- Fill in comparison and casts.
- Where are the local vars stored?

Clicker Question

If you have ResponseCard clicker, channel is **41**.

If you are using ResponseWare, session id is **CS1400U**.

```
int test(data_t a) {
   return a != 0;
}
```

```
testl  %eax,%eax
setne  %al
```

What is **data_t**?

A. `unsigned`

B. `int`

C. `char*`

D. exactly 2 of the above

E. all of A-C

Clicker Question

```
int test(data_t a) {
   return a > 0;
}
```

```
testb  %al, %al
setg  %al
```

What is **data_t**?

A. `char`

B. `unsigned char`

C. `char*`

D. exactly 2 of the above

E. all of A-C

Clicker Question

```
int test(data_t a) {
   return a TEST 0;
}
```

```
testw  %ax, %ax
seta  %al
```

What is **`TEST`**?

A. `&`

B. `==`

C. `<`

D. `>`

E. I don't know

What is **`data_t`**?

A. `short`

B. `unsigned short`

C. `short*`

D. exactly 2 of the above

E. all of A-C

Jump Instructions

- A jump instruction can cause execution to switch to a new position in the program.
  - the jump destination is usually indicated by a label
  - assembler determines the actual addresses of labeled instructions
- `jmp` *label* jumps unconditionally to the indicated *label*.
- `jmp *`*operand* jumps unconditionally to the address read from *operand* (either a register or a memory location).
- *Example*:

```
    xorl %eax,%eax  //what does this do?
    jmp .L1
    movl (%eax),%edx
.L1:
    popl %edx
```

Conditional Jumps

- Other jump instructions either jump to a new position or continue executing at the next instruction depending on some combination of condition codes.
- The names of these jump instructions and the conditions under which they jump match the set instructions.
- *Example*: (let `%edx` contain x and `%eax` contain y)

```
    cmpl %eax,%edx   ;compare x-y
    jl .L1           ;if x<y, jump to L1
    subl %eax,%edx   ;compute x-y
    movl %edx,%eax   ;set x-y as return
    jmp .L2          ;jump to L2
.L1:
    subl %edx,%eax   ;set y-x as return
.L2:
```

Translating Conditional Branches

```
if(test-expr)
    then-stmt
else
    else-stmt
```

C-code template

```
    t = test-expr;
    if(t)
      goto true;
    else-stmt
    goto done;
true:
    then-stmt
done:
```

assembly-code template

- What if there is no *else-stmt*?

Example: Conditional Branches

```
int absdiff(int x, int y) {
  if(x < y)
    return y – x;
  else
    return x – y;
}
```

C code

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

    if(x < y)
      goto less;
    rval = x – y;
    goto done;
  less:
    rval = y - x;
  done
    return rval;
}
```

C code (goto version)

```
  movl 8(%ebp),%edx   ;get x
  movl 12(%ebp),%eax  ;get y
  cmpl %eax,%edx      ;comp x-y
  jl .L3              ;if x<y
  subl %eax,%edx      ;x-y
  movl %edx,%eax      ;ret x-y
  jmp .L5             ;goto done
.L3:
  subl %edx,%eax      ;ret y-x
.L5:
```

`do-while` Loops

```
do
   body-stmt
while(test-expr);
```

C-code template

assembly-code template

```
loop:
   body-stmt
   t = test-expr;
   if(t)
      goto loop;
```

Example: `do-while` Loops

```
int fib_dw(int n) {
  int i = 0;
  int val = 0;
  int nval = 1;

  do {
    int t = val + nval;
    val = nval;
    nval = t;
    i++;
  } while(i < n);

  return val;
}
```

C code

C code (goto version)

```
int fib_dw(int n) {
  // FILL IN

}
```

register	variable	initial val
%ecx	i	0
%esi	n	n
%ebx	val	0
%edx	nval	1
%eax	t	--

```
.L6:
  leal (%edx,%ebx),%eax ;t=...
  movl %edx,%ebx        ;val=nval
  movl %eax,%edx        ;nval=t
  incl %ecx             ;i++
  cmpl %esi,%ecx        ;comp i-n
  jl .L6                ;if i<n
  movl %ebx,%eax        ;ret val
```

`while` Loops

```
while(test-expr)
   body-stmt
```

C-code template

```
loop:
  t = test-expr;
  if(!t)
    goto done;
  body-stmt
  goto loop;
done:
```

assembly-code template

```
  if(!test-expr)
    goto done;
  do
    body-stmt
  while(test-expr);
done:
```

C-code template (do-while style)

```
  t = test-expr;
  if(!t)
    goto done;
loop:
  body-stmt
  t = test-expr;
  if(t)
    goto loop;
done:
```


assembly-code template (do-while style)

Example: `while` Loops

C code

```
int fib_w(int n) {
  int i = 1;
  int val = 1;
  int nval = 1;

  while(i < n) {
    int t = val + nval;
    val = nval;
    nval = t;
    i++;
  }

  return val;
}
```

C code (goto version)

```
int fib_w(int n) {
  // FILL IN

}
```

register	variable	initial val
%edx	nmi	n-1
%ebx	val	1
%ecx	nval	1
%eax	t	--

```
  movl 8(%ebp),%eax   ;get n
  movl $1,%ebx        ;val=1
  movl $1,%ecx        ;nval=1
  cmpl %eax,%ebx    ;comp val-n
  jge .L9             ;if val<n
  leal -1(%eax),%edx ;nmi=n-1
.L10:
  leal (%ecx,%ebx),%eax ;t=...
  movl %ecx,%ebx      ;val=nval
  movl %eax,%ecx      ;nval=t
  decl %edx           ;nmi--
  jnz .L10          ;if nmi!=0
.L9:
```

`for` Loops

```
for(init-expr; test-expr; update-expr)
   body-stmt
```

C-code template

```
   init-expr;
   if(!test-expr)
     goto done;
   do {
     body-stmt
     update-expr;
   } while(test-expr);
done:
```

C-code template (do-while style)

```
   init-expr;
   t = test-expr;
   if(!t)
     goto done;
loop:
   body-stmt
   update-expr;
   t = test-expr;
   if(t)
     goto loop;
done:
```

assembly-code template (do-while style)

Example: `for` Loops

```
int fib_f(int n) {
  int i;
  int val = 1;
  int nval = 1;

  for(i = 1; i < n; i++) {
    int t = val + nval;
    val = nval;
    nval = t;
  }

  return val;
}
```

C code

```
same assembly code as for
fib_w function
```

Exercise: Loops

```
int loop_while(int a, int b) {
  int i = 0;
  int result = a;

  while(i < 256) {
    result += a;
    a -= b;
    i += b;
  }

  return result;
}
```

register	variable	initial val
%eax		
%ebx		
%ecx		
%edx		

```
  movl 8(%ebp),%eax     ;get a
  movl 12(%ebp),%ebx    ;get b
  xorl %ecx,%ecx
  movl %eax,%edx
.L5:
  addl %eax,%edx
  subl %ebx,%eax
  addl %ebx,%ecx
  cmpl $255,%ecx
  jle .L5
  movl %edx,%eax
```

- *test-expr*?
- *body-stmt*?
- compiler optimizations?

`switch` Statements

- Multiway branching based on value of an integer index.
- Useful when dealing with test where there can be a large number of possible outcomes.
  - C code more readable and implementation can be very efficient
- A *jump table* is an array where entry i is the address of a code segment to be executed when switch index $== f(i)$.
  - switch running time is independent of number of cases
- Jump tables are used when the number of cases is more than a few and they span a small range of values.

```
switch(x) {

case 100:
  x *= 13;
  break;

case 102:
  x += 10;

case 103:
  x += 11;
  break;

case 104:
case 106:
  x *= x;
  break;

default:
  x = 0;
}
```

C code

```
  code* jt[] = {A, def, B,
            C, D, def, D};

  unsigned xi = x - 100;

  if(xi > 6)
    goto def;

  goto jt[xi];

A:
  x *= 13;
  goto done;

B:
  x += 10;

C:
  x += 11;
  goto done;

D:
  x *= x;
  goto done;

def:
  x = 0;

done:
```

“extended” C code

```
.section .rodata
  .align 4
.L10
  .long .L4
  .long .L9
  .long .L5
  .long .L6
  .long .L8
  .long .L9
  .long .L8
...
  leal -100(%edx),%eax
  cmpl $6,%eax
  ja .L9
  jmp *.L10(,%eax,4)
.L4:
  leal (%edx,%edx,2),%eax
  leal (%edx,%eax,4),%edx
  jmp .L3
.L5
  addl $10,%edx
.L6
  addl $11,%edx
  jmp .L3
.L8
  imull %edx,%edx
  jmp .L3
.L9
  xorl %edx,%edx
.L3:
```

assembly code

Exercise: `switch` Statements

```
int switch2(int x) {
  int result = 0;

  switch(x) {

    /* OMITTED */
  }

  return result;
}
```

```
.section .rodata
   .align 4
.L11
   .long .L4
   .long .L10
   .long .L5
   .long .L6
   .long .L8
   .long .L8
   .long .L9
...
  movl 8(%ebp),%eax   ;get x
  addl $2,%eax
  cmpl $6,%eax
  ja .L10
  jmp *.L11(,%eax,4)
...
```

- What are the values of the case labels?

- What cases share a label?

gdb Debugger

- The GNU debugger can be used to do run-time evaluation and analysis of machine-level programs.
- Set breakpoints near points of interest.
  - just after function entry, or specific program addresses
  - when breakpoint is reached, control returns to user
  - examine the contents of registers and memory locations
  - single step or proceed to next breakpoint
- See your text (3.11), textbook's web notes, gdb's `help` command, and Google for more info.