System Programming

07. Machine-Level Programming II: Control (ch 3.6)

2019. Fall

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

datalab
data science laboratory

These slides are based on your text book on the slides prepared by Dan Grossman, Ruth Anderson.

Roadmap

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8
Mac

Computer system:

Control Flow

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rax	return value

```
long max(long x, long y)
{
  long max;
  if (x > y) {
    max = x;
  } else {
    max = y;
  }
  return max;
}
```

```
max:
   ???
   movq    %rdi, %rax
   ???
   ???
   movq    %rsi, %rax
   ???
   ret
```

Control Flow

Register	Use(s)
`%rdi`	1st argument (`x`)
`%rsi`	2nd argument (`y`)
`%rax`	return value

```
long max(long x, long y)
{
  long max;
  if (x > y) {
    max = x;
  } else {
    max = y;
  }
  return max;
}
```

Conditional jump

Unconditional jump

```
max:
  if x <= y then jump to else
  movq   %rdi, %rax
  jump to done
else:
  movq   %rsi, %rax
done:
  ret
```

Conditionals and Control Flow

- Conditional branch/*jump*
  - Jump to somewhere else if some *condition* is true, otherwise execute next instruction
- Unconditional branch/*jump*
  - *Always* jump when you get to this instruction

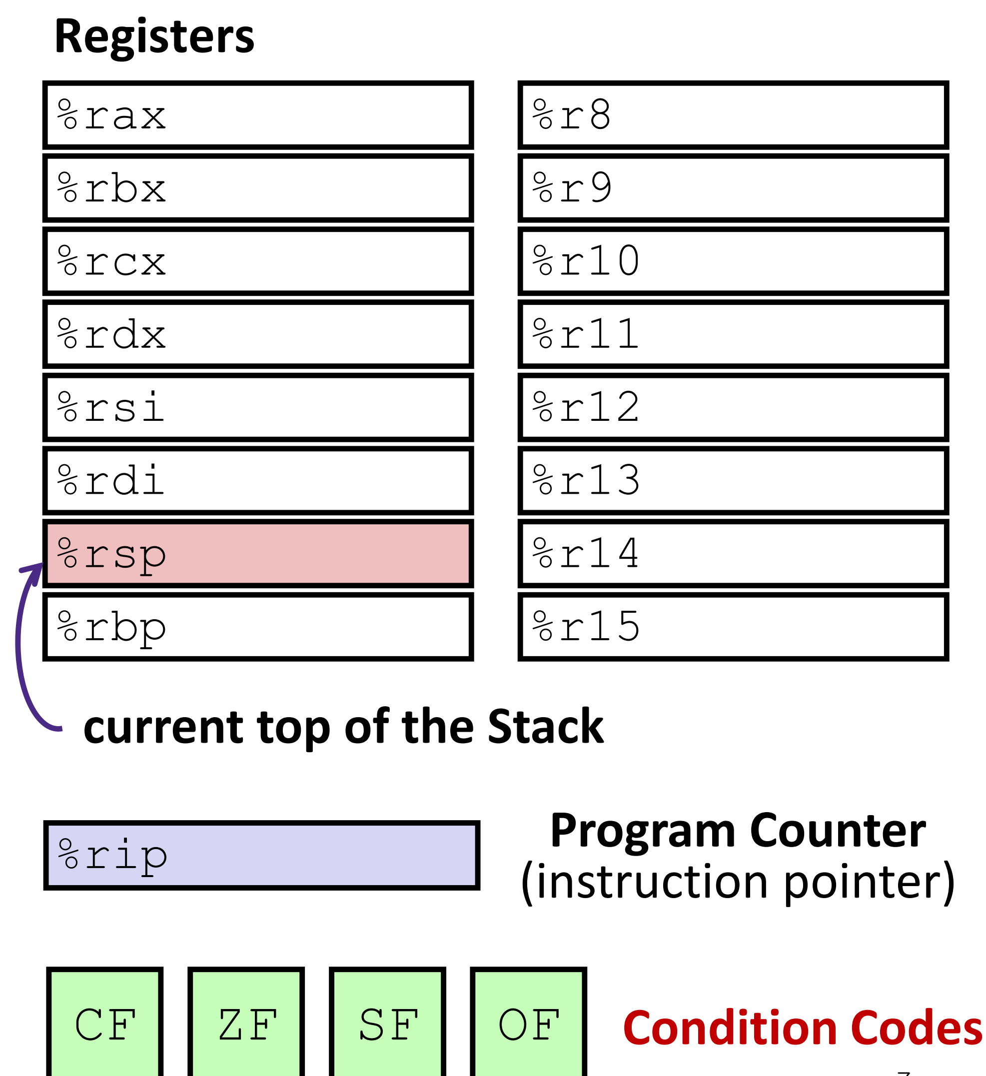
- Together, they can implement most control flow constructs in high-level languages:
  - **`if`** `(`*condition*`)` **`then`** `{…}` **`else`** `{…}`
  - **`while`** `(`*condition*`)` `{…}`
  - **`do`** `{…}` **`while`** `(`*condition*`)`
  - **`for`** `(`*initialization*`;` *condition*`;` *iterative*`)` `{…}`
  - **`switch`** `{…}`

Topics: control flow

- **Condition codes**
- Conditional and unconditional branches
- Loops
- Switches

Processor State (x86-64, partial)

- Information about currently executing program
  - Temporary data (%rax, …)
  - Location of runtime stack (%rsp)
  - Location of current code control point (%rip, …)
  - Status of recent tests (**CF**, **ZF**, **SF**, **OF**)
    - Single bit registers:



Condition Codes (Implicit Setting)

- *Implicitly* set by **arithmetic** operations
  - (think of it as side effects)
  - Example: **`addq`** `src, dst` ↔ `r = d+s`

  - **`CF=1`** if carry out from MSB (unsigned overflow)
  - **`ZF=1`** if `r==0`
  - **`SF=1`** if `r<0` (assuming signed, actually just if MSB is 1)
  - **`OF=1`** if two’s complement (signed) overflow
    `(s>0 && d>0 && r<0)||(s<0 && d<0 && r>=0)`

Not set by `lea` instruction (beware!)

`CF` *Carry Flag* `ZF` *Zero Flag* `SF` *Sign Flag* `OF` *Overflow Flag*

Condition Codes (Explicit Setting: Compare)

- *Explicitly* set by **Compare** instruction
  - **`cmpq`** `src1, src2`
  - **`cmpq`** `a, b` sets flags based on `b-a`, but doesn't store

  - **`CF=1`** if carry out from MSB (used for unsigned comparison)
  - **`ZF=1`** if `a==b`
  - **`SF=1`** if `(b-a)<0` (signed)
  - **`OF=1`** if two's complement (signed) overflow

    `(b>0 && a<0 && (b-a)<0) ||`

    `(b<0 && a>0 && (b-a)>0)`

CF	*Carry Flag*	ZF	*Zero Flag*	SF	*Sign Flag*	OF	*Overflow Flag*

Condition Codes (Explicit Setting: Test)

- *Explicitly* set by **Test** instruction
  - **`testq`** `src2, src1`   like andq src, dst
  - **`testq`** `a, b` sets flags based on `b&a`, but doesn't store
    - Useful to have one of the operands be a ***mask***

- Can't have carry out (**`CF`**) or overflow (**`OF`**)
- **`ZF=1`** if `a&b==0`
- **`SF=1`** if `a&b<0` (signed)

ZF=1 if a&a==0 ⇒ a==0

SF=1 if a&a<0 ⇒ a<0

SF	ZF	what does this say about a?
0	0	a>0
0	1	a==0
1	0	a<0
1	1	shouldn't ever see this!

- Example: `testq %rax, %rax` (a, a)
  - Tells you if (+), 0, or (–) based on ZF and SF

CF	*Carry Flag*	ZF	*Zero Flag*	SF	*Sign Flag*	OF	*Overflow Flag*

Using Condition Codes: Jumping

- `j*` Instructions
  - Jumps to ***target*** (an address) based on condition codes

Instruction	Condition	Description
`jmp` *target*	`1`	Unconditional
`je` *target*	`ZF`	Equal / Zero
`jne` *target*	`~ZF`	Not Equal / Not Zero
`js` *target*	`SF`	Negative
`jns` *target*	`~SF`	Nonnegative
`jg` *target*	`~(SF^OF)&~ZF`	Greater (Signed)
`jge` *target*	`~(SF^OF)`	Greater or Equal (Signed)
`jl` *target*	`(SF^OF)`	Less (Signed)
`jle` *target*	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja` *target*	`~CF&~ZF`	Above (unsigned “>”)
`jb` *target*	`CF`	Below (unsigned “<“)

Using Condition Codes: Setting

- `set*` Instructions
  - Set low-order byte of `dst` to 0 or 1 based on condition codes
  - Does not alter remaining 7 bytes

Instruction	Condition	Description
`sete` *dst*	`ZF`	Equal / Zero
`setne` *dst*	`~ZF`	Not Equal / Not Zero
`sets` *dst*	`SF`	Negative
`setns` *dst*	`~SF`	Nonnegative
`setg` *dst*	`~(SF^OF)&~ZF`	Greater (Signed)
`setge` *dst*	`~(SF^OF)`	Greater or Equal (Signed)
`setl` *dst*	`(SF^OF)`	Less (Signed)
`setle` *dst*	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta` *dst*	`~CF&~ZF`	Above (unsigned “>”)
`setb` *dst*	`CF`	Below (unsigned “<”)

Reminder: x86-64 Integer Registers

- Accessing the low-order byte:

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

Reading Condition Codes

Register	Use(s)
`%rdi`	1st argument (`x`)
`%rsi`	2nd argument (`y`)
`%rax`	return value

- `set*` Instructions
  - Set a low-order byte to 0 or 1 based on condition codes
  - Operand is byte register (e.g. `al`, `dl`) or a byte in memory
  - Do not alter remaining bytes in register
    - Typically use `movzbl` (zero-extended `mov`) to finish job

```
int gt(long x, long y)
{
  return x > y;
}
```

```
cmpq   %rsi, %rdi    #
setg   %al           #
movzbl %al, %eax     #
ret
```

Reading Condition Codes

Register	Use(s)
`%rdi`	1st argument (`x`)
`%rsi`	2nd argument (`y`)
`%rax`	return value

- `set*` Instructions
  - Set a low-order byte to 0 or 1 based on condition codes
  - Operand is byte register (e.g. `al`, `dl`) or a byte in memory
  - Do not alter remaining bytes in register
    - Typically use `movzbl` (zero-extended `mov`) to finish job

```c
int gt(long x, long y)
{
  return x > y;
}
```

```
cmpq   %rsi, %rdi    # Compare x:y
setg   %al           # Set when > 0
movzbl %al, %eax     # Zero rest of %rax
ret
```

Aside: `movz` and `movs`

`movz__` *src, regDest* *Move with zero extension*

`movs__` *src, regDest* *Move with sign extension*

- Copy from a *smaller* source value to a *larger* destination
- Source can be memory or register; Destination *must* be a register
- Fill remaining bits of dest with **zero** (`movz`) or **sign bit** (`movs`)

`movzSD` / `movsSD`:

S – size of source (**`b`** = 1 byte, **`w`** = 2)

D – size of dest (**`w`** = 2 bytes, **`l`** = 4, **`q`** = 8)

Example:

`movzbq %al, %rbx`

0x??	0x??	0x??	0x??	0x??	0x??	0x??	0xFF	←%rax
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0xFF	←%rbx

Aside: `movz` and `movs`

`movz__` *src, regDest* *Move with zero extension*

`movs__` *src, regDest* *Move with sign extension*

- Copy from a *smaller* source value to a *larger* destination
- Source can be memory or register; Destination *must* be a register
- Fill remaining bits of dest with **zero** (`mov`**z**) or **sign bit** (`mov`**s**)

`movz`*SD* / **`movs`*SD***:

S – size of source (**b** = 1 byte, **w** = 2)

D – size of dest (**w** = 2 bytes, **l** = 4, **q** = 8)

Note: In x86-64, *any instruction* that generates a 32-bit (long word) value for a register also sets the high-order portion of the register to 0. Good example on p. 184 in the textbook.

Example:

`movsbl (%rax), %ebx`

0x00	0x00	0x7F	0xFF	0xC6	0x1F	0xA4	0xE8	←%rax

...	0x??	0x??	0x80	0x??	0x??	0x??	...	← MEM

0x00	0x00	0x00	0x00	0xFF	0xFF	0xFF	0x80	←%rbx

Copy 1 byte from memory into 8-byte register & sign extend it

Topics: control flow

- Condition codes
- **Conditional and unconditional branches**
- Loops
- Switches

Choosing instructions for conditionals (1)

- All arithmetic instructions set condition flags based on result of operation (op)
  - Conditionals are comparisons against 0
- Come in instruction *pairs*

```
      addq 5, (p)
je:    *p+5 == 0
jne:   *p+5 != 0
jg:    *p+5 >  0
jl:    *p+5 <  0
```

```
      orq a, b
je:    b|a == 0
jne:   b|a != 0
jg:    b|a >  0
jl:    b|a <  0
```

		(op) s, d
je	"Equal"	`d (op) s == 0`
jne	"Not equal"	`d (op) s != 0`
js	"Sign" (negative)	`d (op) s < 0`
jns	(non-negative)	`d (op) s >= 0`
jg	"Greater"	`d (op) s > 0`
jge	"Greater or equal"	`d (op) s >= 0`
jl	"Less"	`d (op) s < 0`
jle	"Less or equal"	`d (op) s <= 0`
ja	"Above" (unsigned >)	`d (op) s > 0U`
jb	"Below" (unsigned <)	`d (op) s < 0U`

Choosing instructions for conditionals (2)

- Reminder: `cmp` is like `sub`, `test` is like `and`
  - Result is not stored anywhere

		cmp a,b	**test a,b**
je	"Equal"	`b == a`	`b&a == 0`
jne	"Not equal"	`b != a`	`b&a != 0`
js	"Sign" (negative)	`b-a < 0`	`b&a < 0`
jns	(non-negative)	`b-a >=0`	`b&a >= 0`
jg	"Greater"	`b > a`	`b&a > 0`
jge	"Greater or equal"	`b >= a`	`b&a >= 0`
jl	"Less"	`b < a`	`b&a < 0`
jle	"Less or equal"	`b <= a`	`b&a <= 0`
ja	"Above" (unsigned >)	`b > a`	`b&a > 0U`
jb	"Below" (unsigned <)	`b < a`	`b&a < 0U`

```
    cmpq 5, (p)
je:   *p == 5
jne:  *p != 5
jg:   *p >  5
jl:   *p <  5
```

```
    testq a, a
je:    a == 0
jne:   a != 0
jg:    a >  0
jl:    a <  0
```

```
    testb a, 0x1
je:    a_LSB == 0
jne:   a_LSB == 1
```

Choosing instructions for conditionals (3)

		`cmp a,b`	`test a,b`
je	"Equal"	`b == a`	`b&a == 0`
jne	"Not equal"	`b != a`	`b&a != 0`
js	"Sign" (negative)	`b-a < 0`	`b&a < 0`
jns	(non-negative)	`b-a >=0`	`b&a >= 0`
jg	"Greater"	`b > a`	`b&a > 0`
jge	"Greater or equal"	`b >= a`	`b&a >= 0`
jl	"Less"	`b < a`	`b&a < 0`
jle	"Less or equal"	`b <= a`	`b&a <= 0`
ja	"Above" (unsigned >)	`b > a`	`b&a > 0U`
jb	"Below" (unsigned <)	`b < a`	`b&a < 0U`

Register	Use(s)
`%rdi`	argument `x`
`%rsi`	argument `y`
`%rax`	return value

```
if (x < 3) {
  return 1;
}
return 2;
```

```
  cmpq $3, %rdi
  jge T2
T1: # x < 3:
  movq $1, %rax
  ret
T2: # !(x < 3):
  movq $2, %rax
  ret
```

Question

Register	Use(s)
`%rdi`	1st argument (`x`)
`%rsi`	2nd argument (`y`)
`%rax`	return value

A.
```
cmpq  %rsi, %rdi
jle   .L4
```

B.
```
cmpq  %rsi, %rdi
jg    .L4
```

C.
```
testq %rsi, %rdi
jle   .L4
```

D.
```
testq %rsi, %rdi
jg    .L4
```

E. We're lost…

```
long absdiff(long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:

   ____________________

   ____________________
                          # x > y:
   movq     %rdi, %rax
   subq     %rsi, %rax
   ret
.L4:                      # x <= y:
   movq     %rsi, %rax
   subq     %rdi, %rax
   ret
```

Choosing instructions for conditionals (4)

		`cmp a,b`	`test a,b`
`je`	"Equal"	`b == a`	`b&a == 0`
`jne`	"Not equal"	`b != a`	`b&a != 0`
`js`	"Sign" (negative)	`b-a < 0`	`b&a < 0`
`jns`	(non-negative)	`b-a >=0`	`b&a >= 0`
`jg`	"Greater"	`b > a`	`b&a > 0`
`jge`	"Greater or equal"	`b >= a`	`b&a >= 0`
`jl`	"Less"	`b < a`	`b&a < 0`
`jle`	"Less or equal"	`b <= a`	`b&a <= 0`
`ja`	"Above" (unsigned >)	`b > a`	`b&a > 0U`
`jb`	"Below" (unsigned <)	`b < a`	`b&a < 0U`

```
if (x < 3 && x == y) {
  return 1;
} else {
  return 2;
}
```

```
  cmpq $3, %rdi
  setl %al
  cmpq %rsi, %rdi
  sete %bl
  testb %al, %bl
  je T2
T1: # x < 3 && x == y:
  movq $1, %rax
  ret
T2: # else
  movq $2, %rax
  ret
```

Choosing instructions for conditionals (5)

		`cmp a,b`	`test a,b`
`je`	"Equal"	`b == a`	`b&a == 0`
`jne`	"Not equal"	`b != a`	`b&a != 0`
`js`	"Sign" (negative)	`b-a < 0`	`b&a < 0`
`jns`	(non-negative)	`b-a >=0`	`b&a >= 0`
`jg`	"Greater"	`b > a`	`b&a > 0`
`jge`	"Greater or equal"	`b >= a`	`b&a >= 0`
`jl`	"Less"	`b < a`	`b&a < 0`
`jle`	"Less or equal"	`b <= a`	`b&a <= 0`
`ja`	"Above" (unsigned >)	`b > a`	`b&a > 0U`
`jb`	"Below" (unsigned <)	`b < a`	`b&a < 0U`

- https://godbolt.org/g/Ovh3jN

```
if (x < 3 && x == y) {
  return 1;
} else {
  return 2;
}
```

```
  cmpq $3, %rdi
  setl %al
  cmpq %rsi, %rdi
  sete %bl
  testb %al, %bl
  je T2
T1: # x < 3 && x == y:
  movq $1, %rax
  ret
T2: # else
  movq $2, %rax
  ret
```

Conditional Branch Example (Old Style)

- Generation

```
$ gcc –Og -S –fno-if-conversion control.c
```

```
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
   cmpq    %rsi, %rdi   # x:y
   jle     .L4
   movq    %rdi, %rax
   subq    %rsi, %rax
   ret
.L4:       # x <= y
   movq    %rsi, %rax
   subq    %rdi, %rax
   ret
```

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rax`	Return value

Expressing with Goto Code

- C allows **`goto`** statement
- Jump to position designated by label

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j
  (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

General Conditional Expression Translation (Using Branches)

C Code

```
val = Test ? Then-Expr : Else-Expr;
```

```
if (Test)
   val = Then-Expr;
else
   val = Else-Expr;
```

Example:

```
result = x>y ? x-y : y-x;
```

Goto Version

```
   ntest = !Test;
   if (ntest) goto Else;
   val = Then_Expr;
   goto Done;
Else:
   val = Else_Expr;
Done:
   . . .
```

- Ternary operator `?:`
- *Test* is expression returning integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one

Conditional Move

- Conditional Move Instructions: **`cmov`*`C`*** `src, dst`
  - Move value from `src` to `dst` if condition ***C*** holds
  - `if(`*Test*`)` *`Dest`* ← *`Src`*
  - GCC tries to use them (but only when known to be **safe**)
- Why is this useful?
  - Branches are very disruptive to instruction flow through *pipelines*
  - Conditional moves do not require control transfer

```
long absdiff(long x, long y)
{
  return x>y ? x-y : y-x;
}
```

```
absdiff:
  movq    %rdi, %rax # x
  subq    %rsi, %rax # result=x-y
  movq    %rsi, %rdx
  subq    %rdi, %rdx # else_val=y-x
  cmpq    %rsi, %rdi # x:y
  cmovle  %rdx, %rax # if <=,
  ret                #    result=else_val
```

Using Conditional Moves

- Conditional Move Instructions
  - **cmov*C*** src, dest
  - Move value from src to dest if condition ***C*** holds
  - Instruction supports:

    if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be **safe**
- Why is this useful?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

C Code

```
val = Test
  ? Then_Expr
  : Else_Expr;
```

"Goto" Version

```
result = Then_Expr;
else_val = Else_Expr;
nt = !Test;
if (nt) result = else_val;
return result;
```

Conditional Move Example

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rax`	Return value

```
absdiff:
   movq    %rdi, %rax    # x
   subq    %rsi, %rax    # result = x-y
   movq    %rsi, %rdx
   subq    %rdi, %rdx    # else_val = y-x
   cmpq    %rsi, %rdi    # x:y
   cmovle  %rdx, %rax    # if <=, result = else_val
   ret
```

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations a re 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
- May have undesirable effects

Summary

- Control flow in x86 determined by status of Condition Codes
  - Showed Carry, Zero, Sign, and Overflow, though others exist
  - Set flags with arithmetic instructions (implicit) or Compare and Test (explicit)
  - Set instructions read out flag values
  - Jump instructions use flag values to determine next instruction to execute

Q&A