Machine-Level Programming V: Advanced Topics & Processor Arch: ISA& Logic

严思明
dantes@pku.edu.cn

1.

Memory layout

I. Text Segment

- Executable machine instructions
- Read-only

II. Data (initialized and uninitialized)

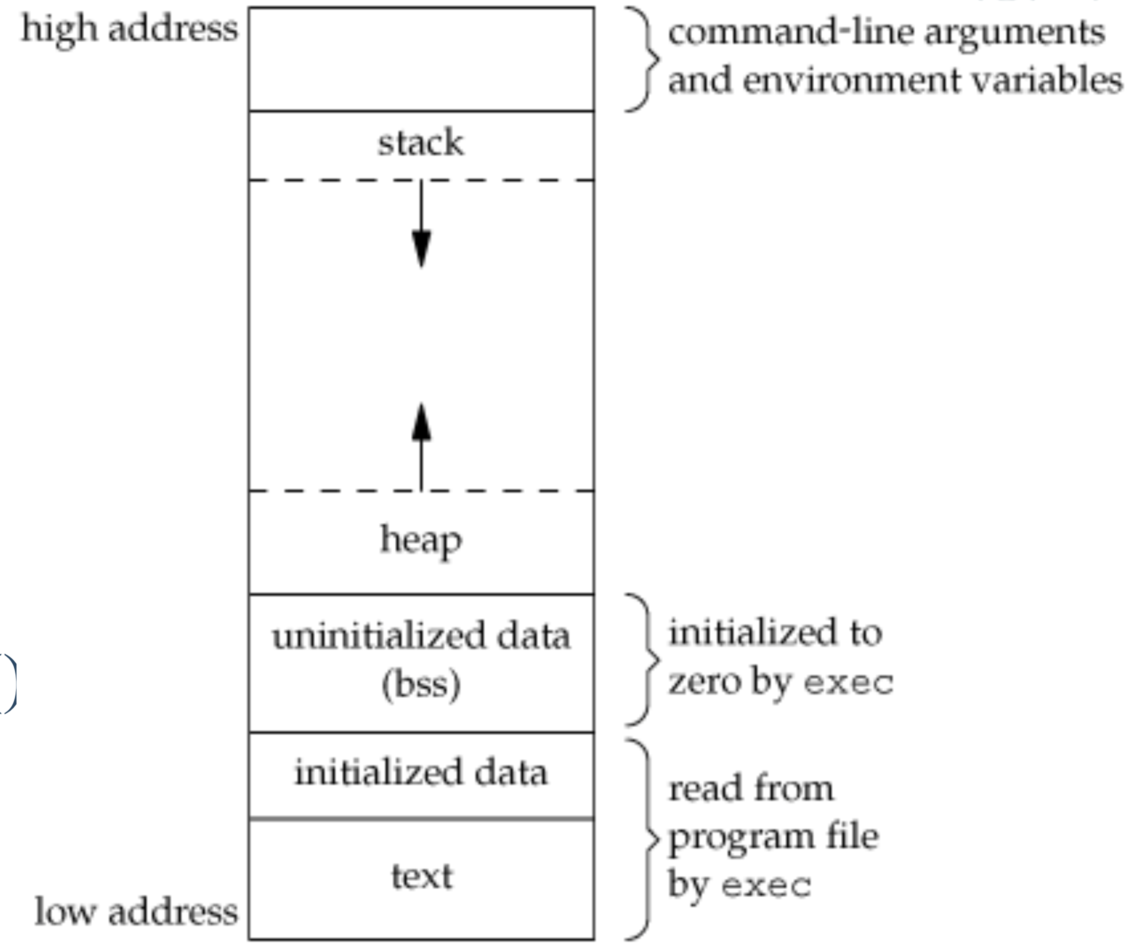
- Statically allocated data
- global, static

III. Stack

- address to return
- local variables

IV. Heap

- Dynamically allocated
- When call malloc(), calloc()
- new()



2.

Buffer overflow

Implementation of Unix function gets()

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

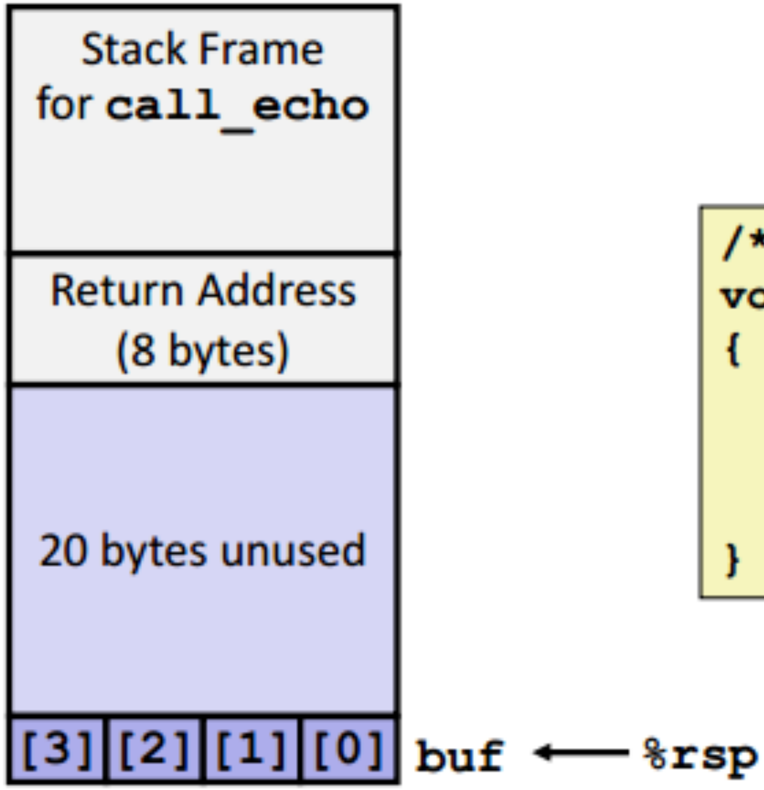
```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
Segmentation Fault
```

Buffer Overflow Stack

Before call to gets




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

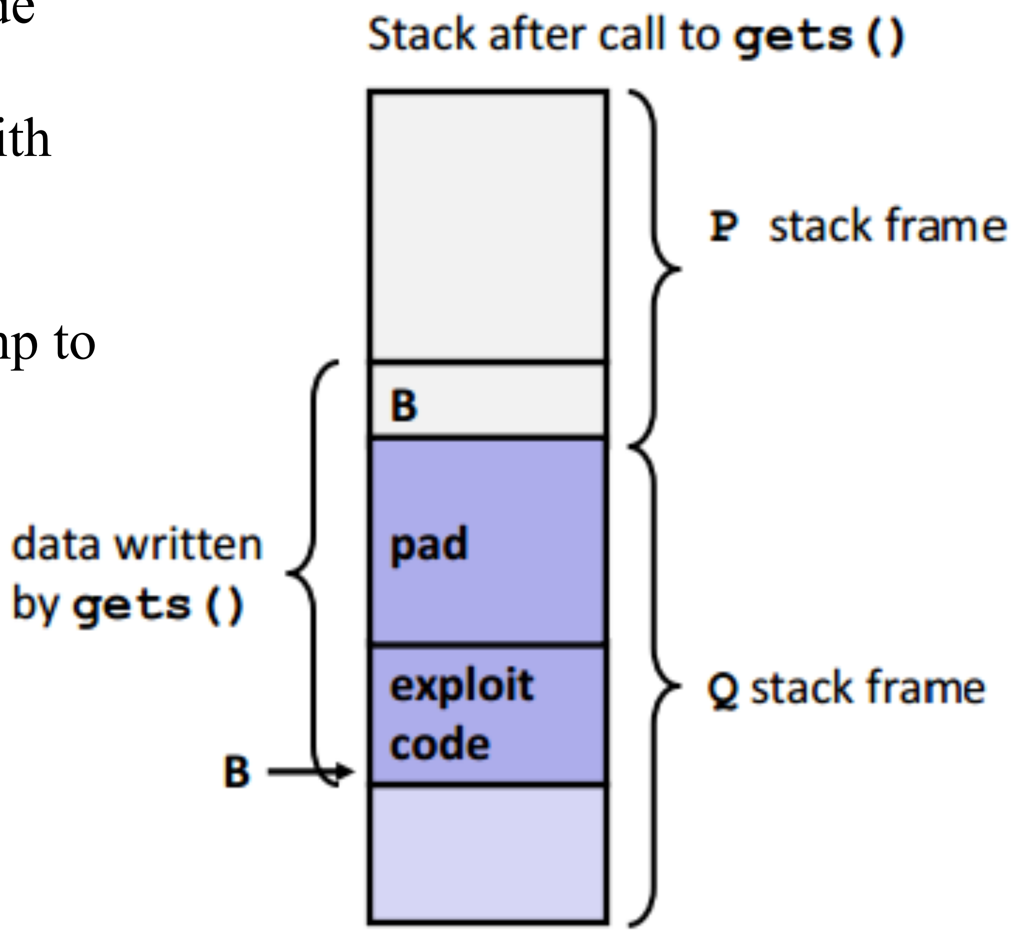
```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

Code Injection Attacks

1.Input string contains byte representation of executable code

2.Overwrite return address A with address of buffer B

3.When Q executes ret, will jump to exploit code



What to do about buffer overflow attacks?

The war between hackers and security engineers

Engineers: Avoid overflow vulnerabilities

- fgets instead of gets

```
char *fgets(char *buf, int bufsize, FILE *stream);
```

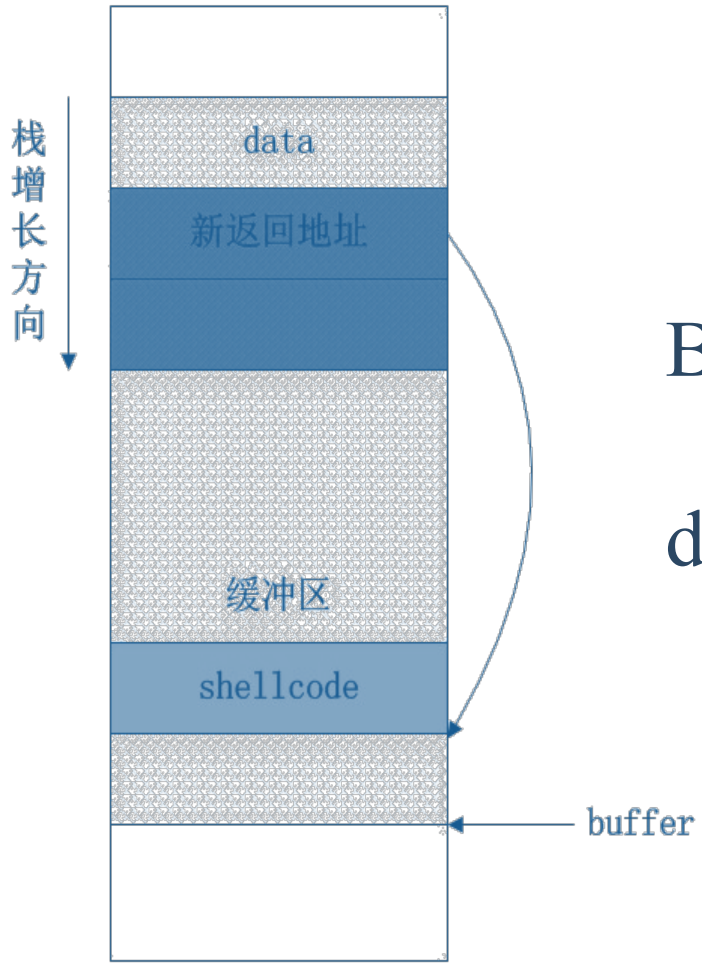
- strncpy instead of strcpy

System-Level Protections can help

Randomized stack offsets

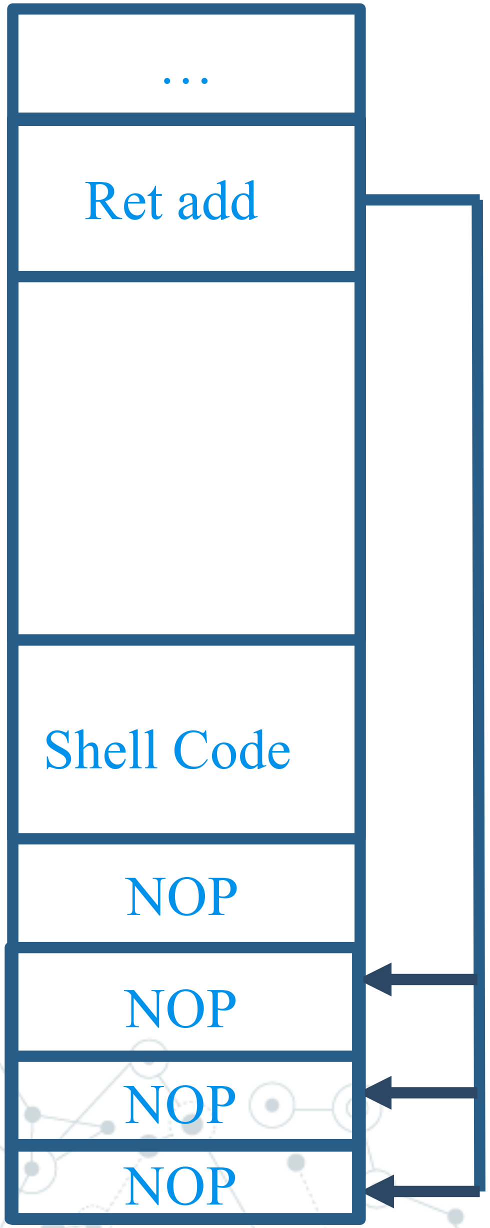
Non-executable code segments

Engineers: Randomized stack offsets



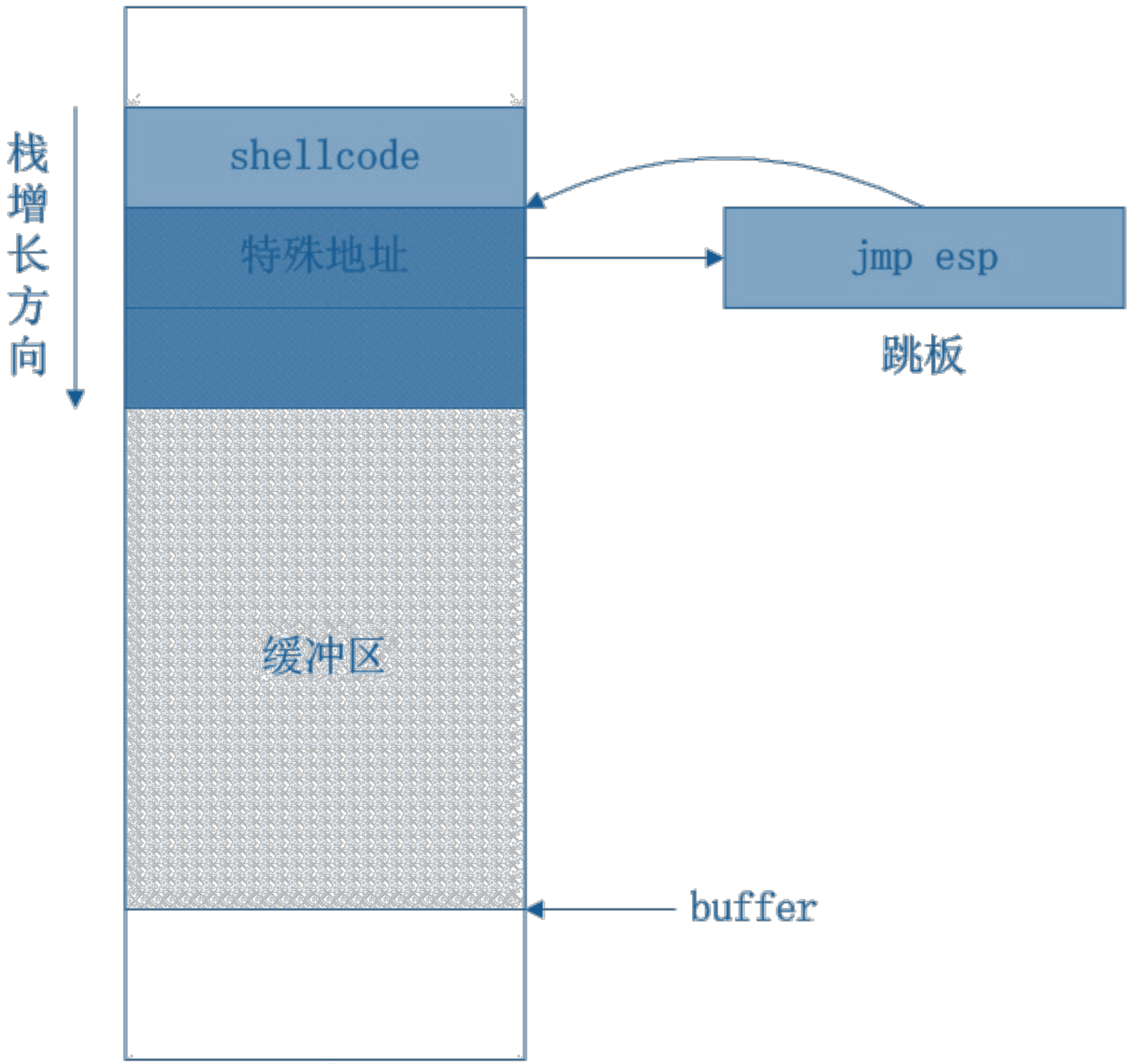
By randomization, the attacker don’t know where the shellcode is.

Hacker1: NOP Sled

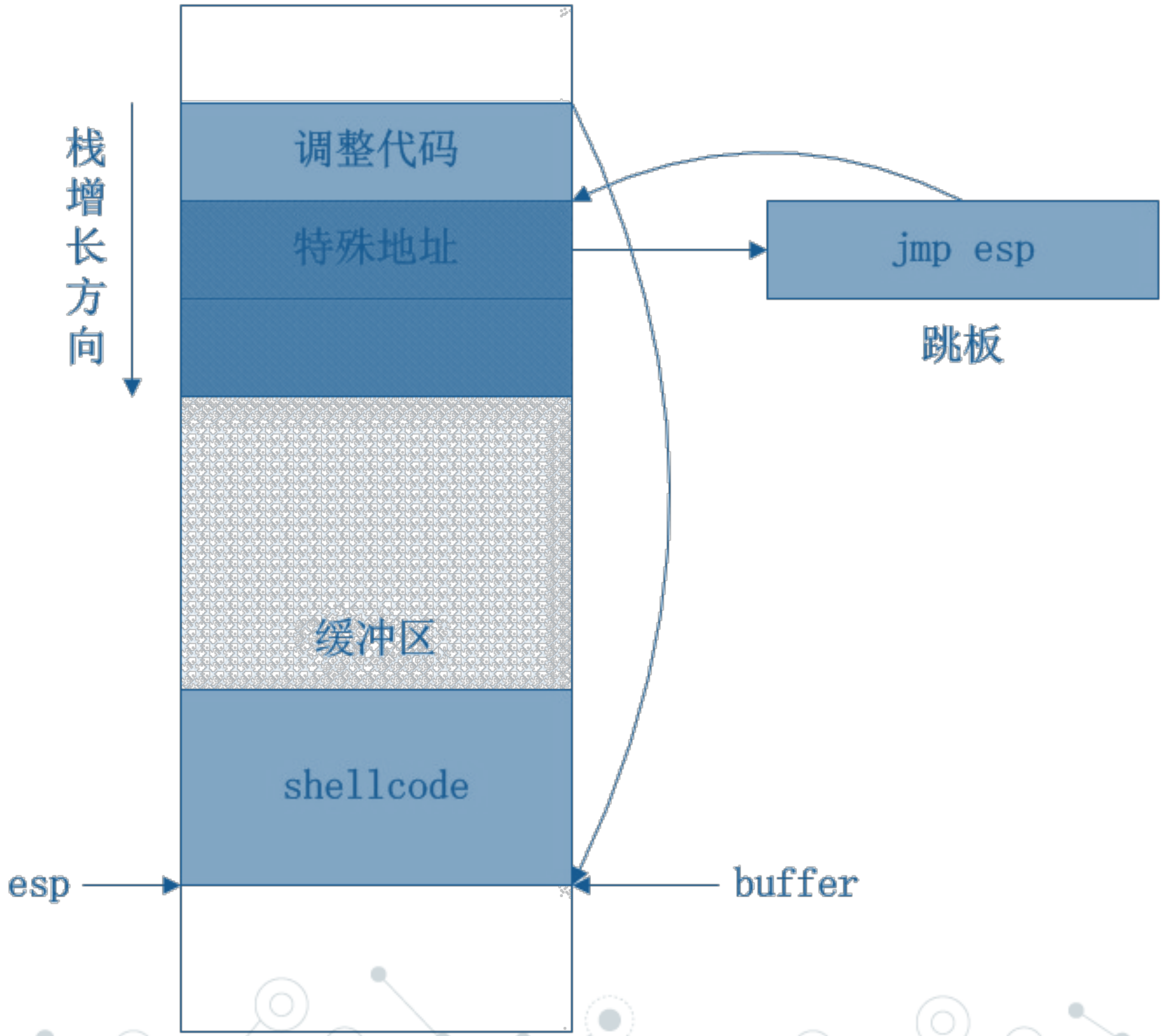


NOP is an instruction that do nothing…

Hacker2: jmp esp



Hacker3: Advanced Attack

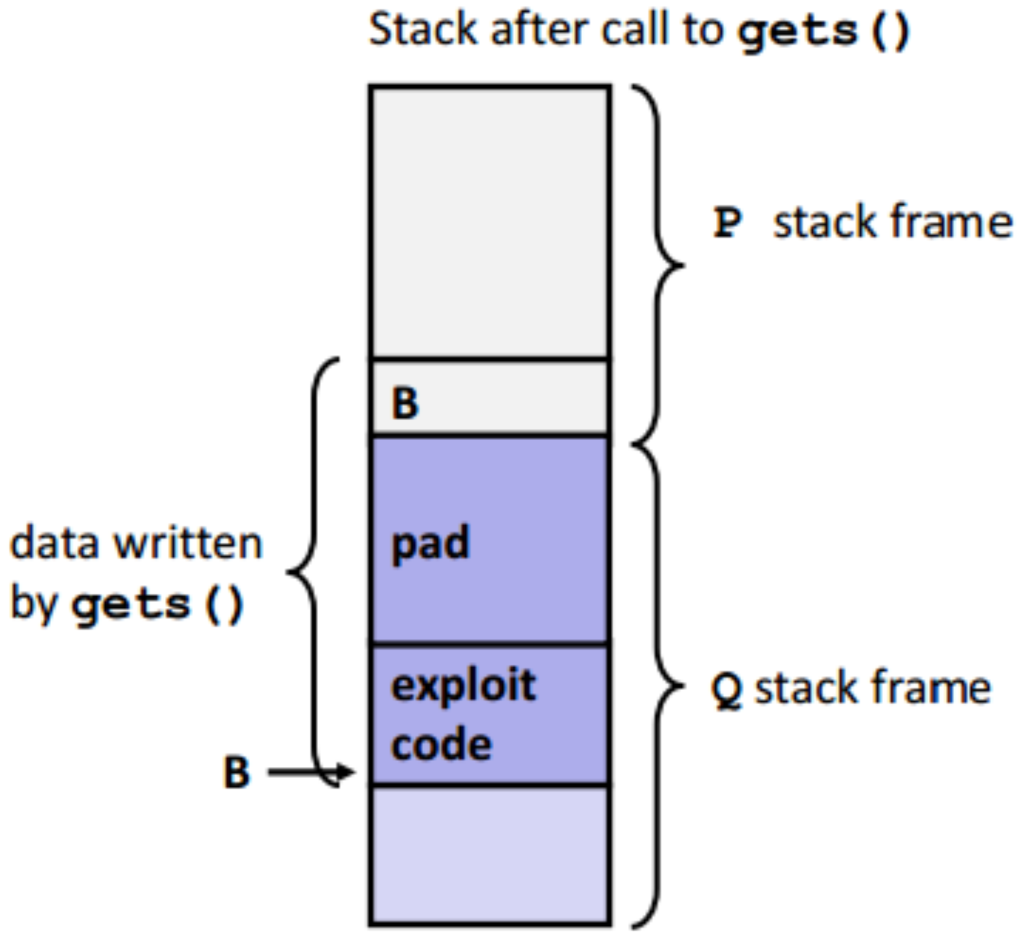


Engineers: Non-executable code segments

In traditional x86, can mark region of memory as either “read-only” or “writeable”

X86-64 added explicit “execute” permission

Stack marked as non-executable



Engineers: Stack Canaries


```
echo:
    . . .
    movq     %fs:40, %rax   # Get canary
    movq     %rax, 8(%rsp)  # Place on stack
    xorl     %eax, %eax     # Erase canary
    . . .
```

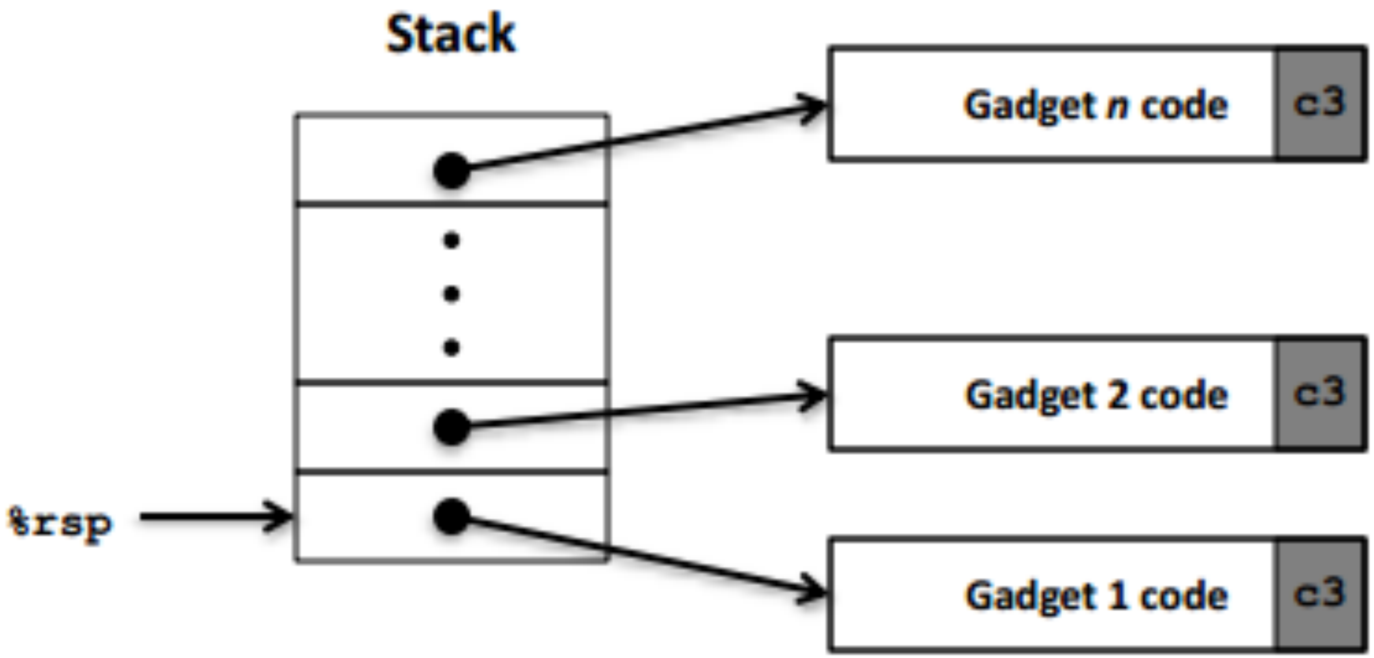
Hacker’s Final Attack: Return-Oriented Programming Attacks

Alternative Strategy

- Use existing code
  - E.g., library code from stdlib
- String together fragments to achieve overall
- desired outcome

Does not overcome stack canaries

ROP Execution

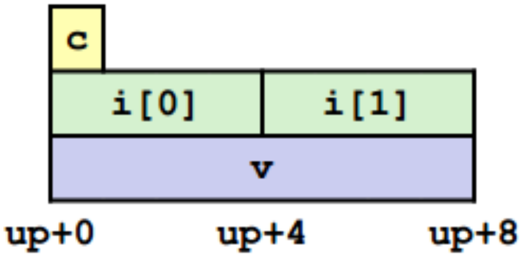


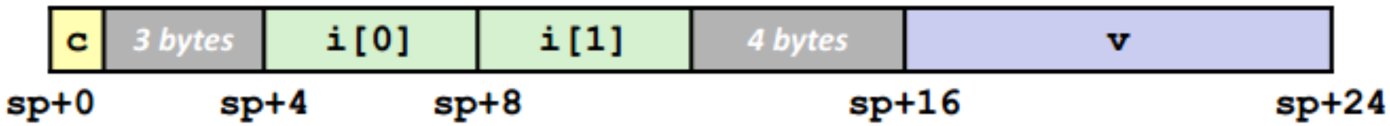
3.

Unions

Union Allocation

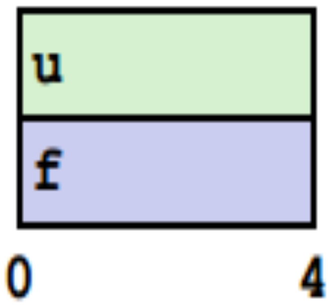
- Allocate according to largest element
- Can only use one field at a time





Using Union to Acess Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

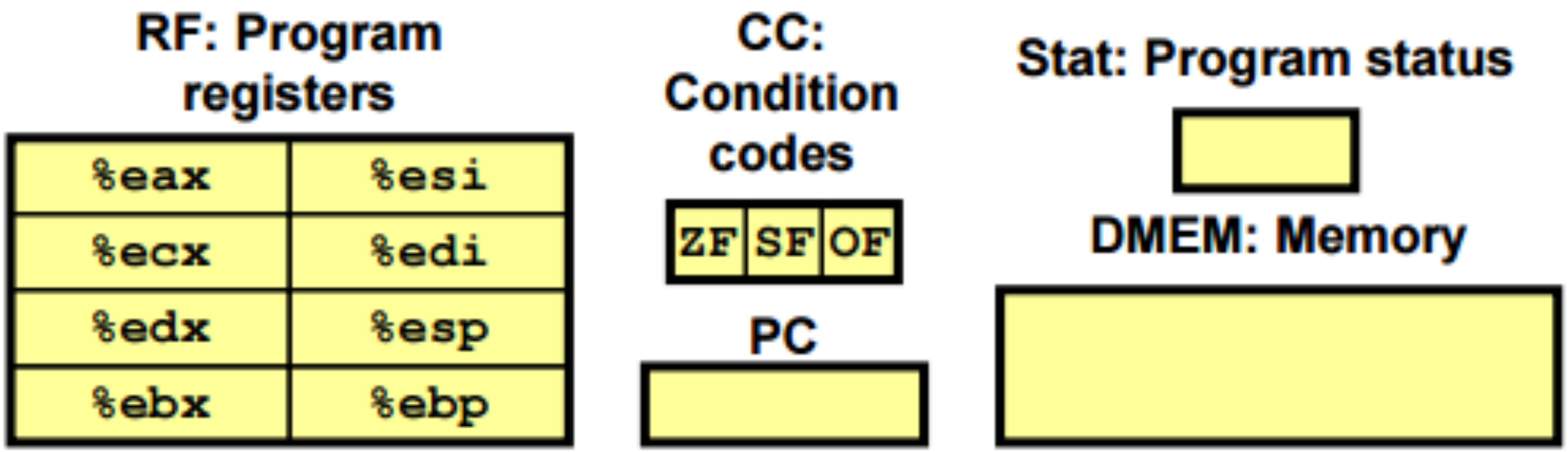
Processor Arch：ISA& Logic

1.

Instruction Set Architecture

Y86 Processor State

- Program Registers
  - Same 8 as with IA32. Each 32 bits
- Condition Codes
  - Single-bit flags set by arithmetic or logical instruction



Y86 Instruction Set

Byte	0		1		2	3	4	5
`halt`	0	0						
`nop`	1	0						
`rrmovl` **rA, rB**	2	0	rA	rB				
`irmovl` **V, rB**	3	0	F	rB	V			
`rmmovl` **rA, D(rB)**	4	0	rA	rB	D			
`mrmovl` **D(rB), rA**	5	0	rA	rB	D			
`OPl` **rA, rB**	6	fn	rA	rB				
`jXX` **Dest**	7	fn	Dest					
`cmovXX` **rA, rB**	2	fn	rA	rB				
`call` **Dest**	8	0	Dest					
`ret`	9	0						
`pushl` **rA**	A	0	rA	F				
`popl` **rA**	B	0	rA	F				

Y86 Instruction Set

Byte	0	1	2	3	4	5
`halt`	0 0					
`nop`	1 0					
`cmovXX rA, rB`	2 fn	rA rB				
`irmovl V, rB`	3 0	F rB	V			
`rmmovl rA, D(rB)`	4 0	rA rB	D			
`mrmovl D(rB), rA`	5 0	rA rB	D			
`OPl rA, rB`	6 fn	rA rB				
`jXX Dest`	7 fn	Dest				
`call Dest`	8 0	Dest				
`ret`	9 0					
`pushl rA`	A 0	rA F				
`popl rA`	B 0	rA F				

`rrmovl`	2 0
`cmovle`	2 1
`cmovl`	2 2
`cmove`	2 3
`cmovne`	2 4
`cmovge`	2 5
`cmovg`	2 6

Encoding Registers

数字	寄存器名字
0	%eax
1	%ecx
2	%edx
3	%ebx
4	%esp
5	%ebp
6	%esi
7	%edi
F	无寄存器

Question:

rmmovl % esp, 0x12345(%edx)的字节编码？（rmmovl的第一个字节为 40）

Statue Conditions

Mnemonic	Code
AOK	1

Normal operation

Mnemonic	Code
HLT	2

Halt instruction encountered

Mnemonic	Code
ADR	3

Bad address (eitherinstruction or data) Encountered

Mnemonic	Code
INS	4

Invalid instruction encountered

Writing Y86 Code

- Try to Use C Compiler as Much as Possible
  - Write code in C
  - Compile for IA32 with gcc –O1 –S
  - Transliterate into Y86
- Coding Example

Y86 Code Generation Example

First Try

- Write typical array code

```
/* Find number of elements in
   null-terminated list */
int len1(int a[])
{
  int len;
  for (len = 0; a[len];
len++)
      ;
  return len;
}
```

- Compile with `gcc34 –O1 -S`

Problem

- Hard to do array indexing on Y86
  - Since don't have scaled addressing modes

```
L5:
   incl   %eax
   cmpl   $0, (%edx,%eax,4)
   jne L5
```

Y86 Code Generation Example #2

Second Try

- Write with pointer code

```
/* Find number of elements in
   null-terminated list */
int len2(int a[])
{
  int len = 0;
  while (*a++)
      len++;
  return len;
}
```

- Compile with `gcc34 -O1 -S`

Result

- Don't need to do indexed addressing

```
.L11:
   incl      %ecx
   movl      (%edx), %eax
   addl      $4, %edx
   testl     %eax, %eax
   jne .L11
```

Y86 Code Generation Example #3

IA32 Code

- Setup

```
len2:
   pushl %ebp
   movl %esp, %ebp




   movl 8(%ebp), %edx
   movl $0, %ecx
   movl (%edx), %eax
   addl  $4, %edx
   testl %eax, %eax
   je  .L13
```

Y86 Code

- Setup

```
len2:
   pushl %ebp          # Save %ebp
   rrmovl %esp, %ebp # New FP
   pushl %esi          # Save
   irmovl $4, %esi     # Constant 4
   pushl %edi          # Save
   irmovl $1, %edi     # Constant 1
   mrmovl 8(%ebp), %edx  # Get a
   irmovl $0, %ecx       # len = 0
   mrmovl (%edx), %eax   # Get *a
   addl %esi, %edx       # a++
   andl %eax, %eax       # Test *a
   je Done    # If zero, goto Done
```

CISC & RISC

◎CISC: Complex Instruction Set Computer
-x86
-Stack-Oriented instruction set
◎RISC: Reduced Instruction Set Computer
-IBM
-Register-oriented instruction set

2.
Logic Design

Overview of Logic Design

- Fundamental Hardware Requirements

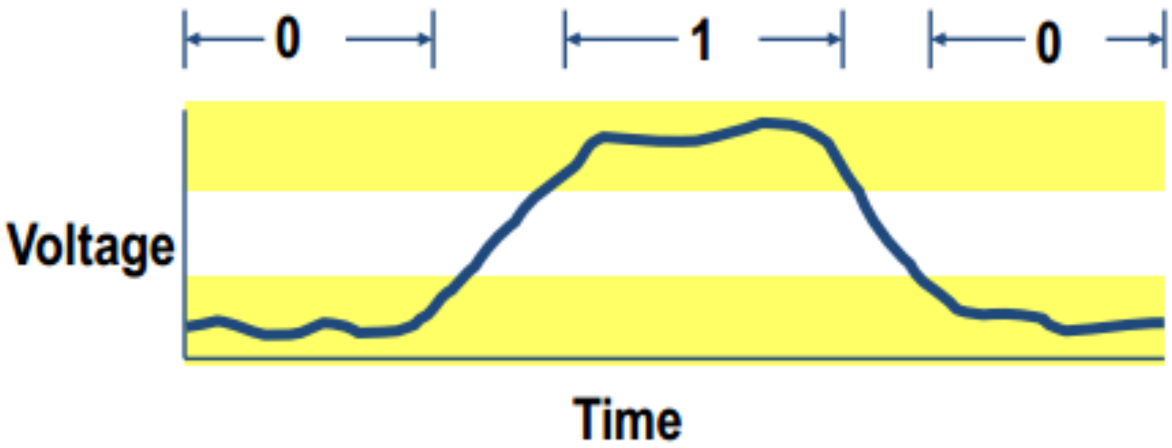
  -Communication

    -How to get values from one place to another

  -Computation

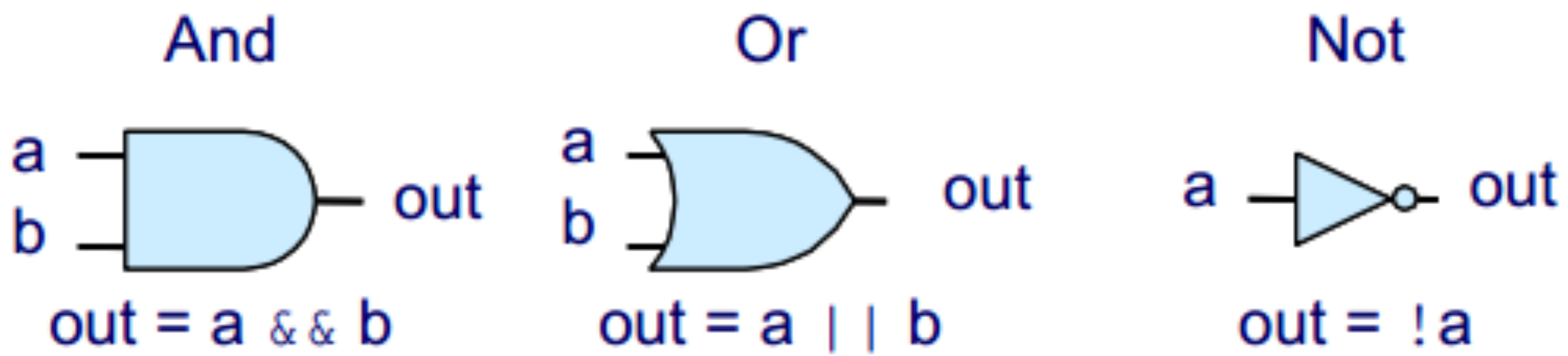
  -Storage

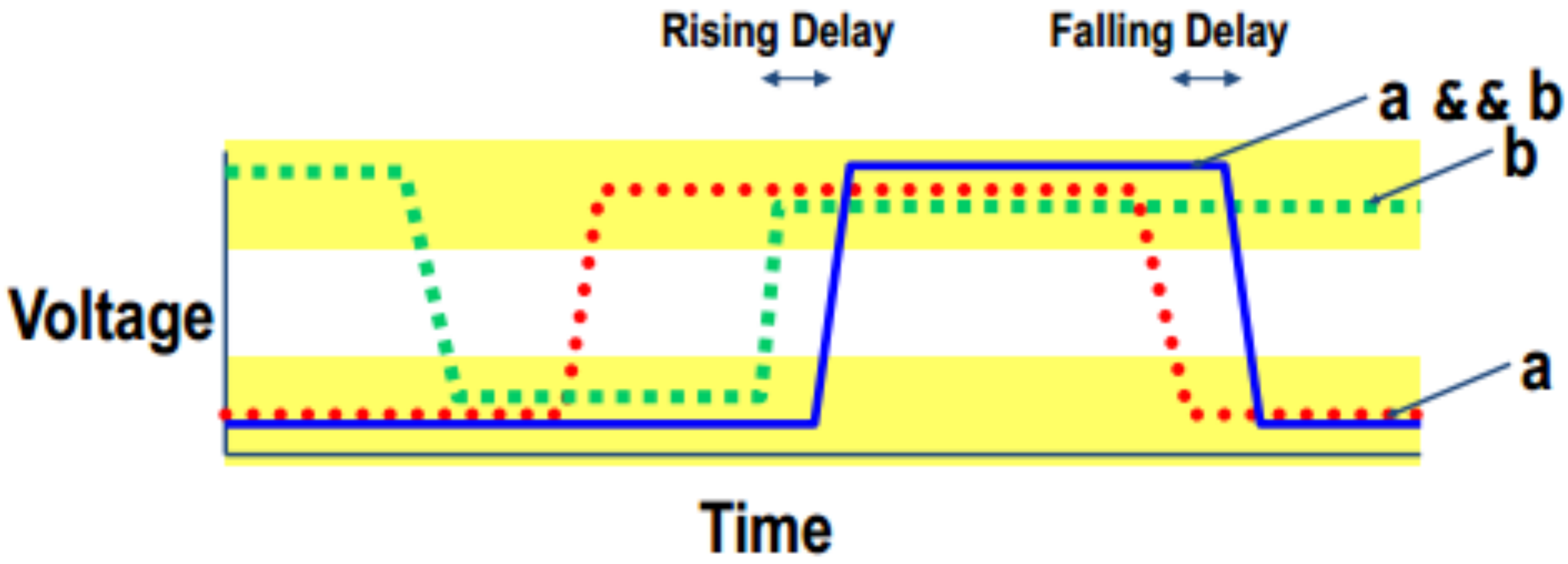
Digital Signals



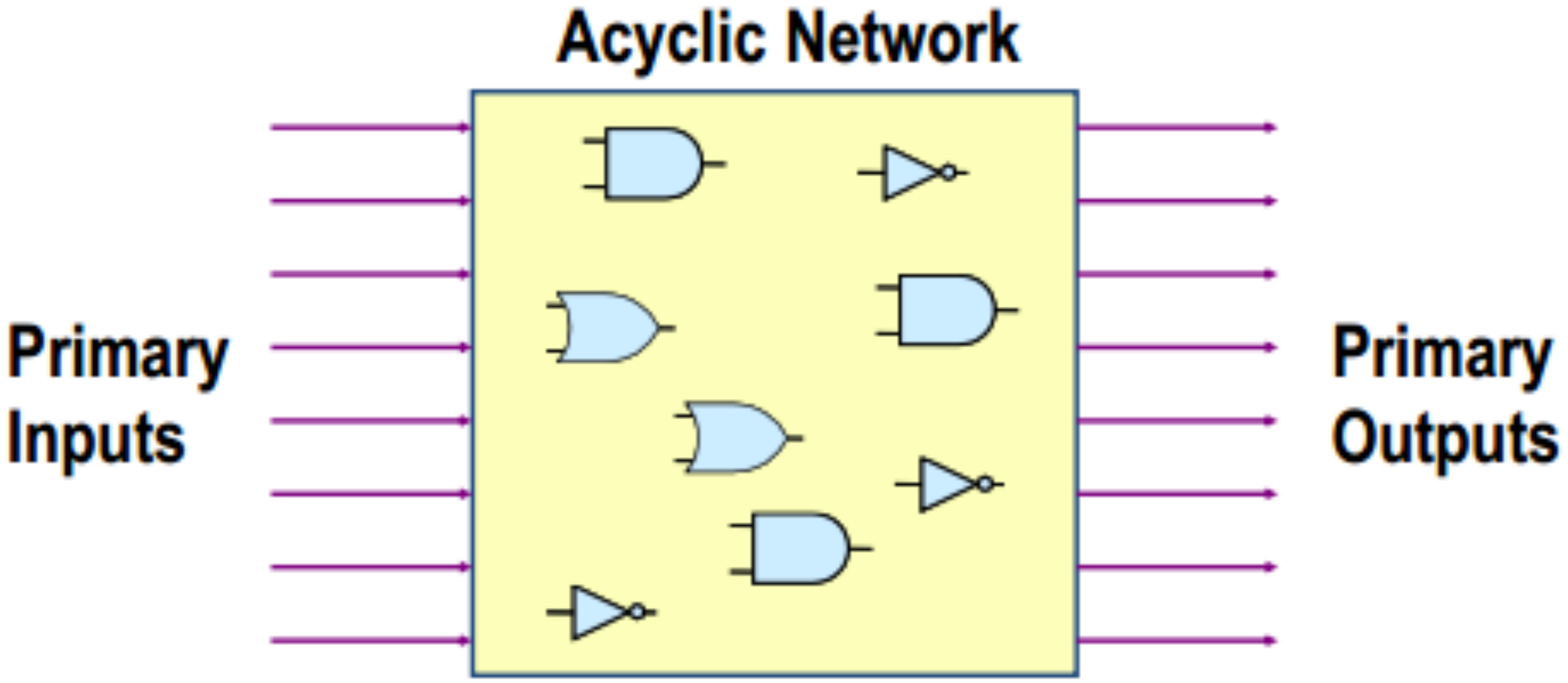
- Use voltage thresholds to extract discrete values from continuous signal
- Simplest version: 1-bit signal
  - Either high range (1) or low range (0)
  - With guard range between them
- Not strongly affected by noise or low quality circuit elements
  - Can make circuits simple, small, and fast

Computing with Logic Gates

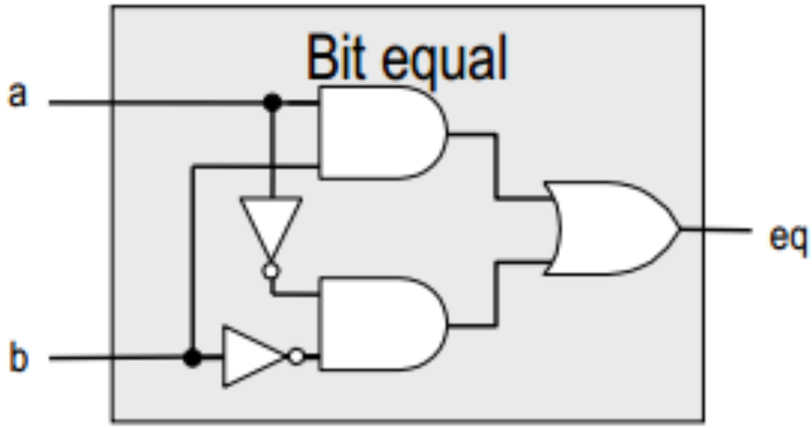




Combinational Circuits



Bit Equality



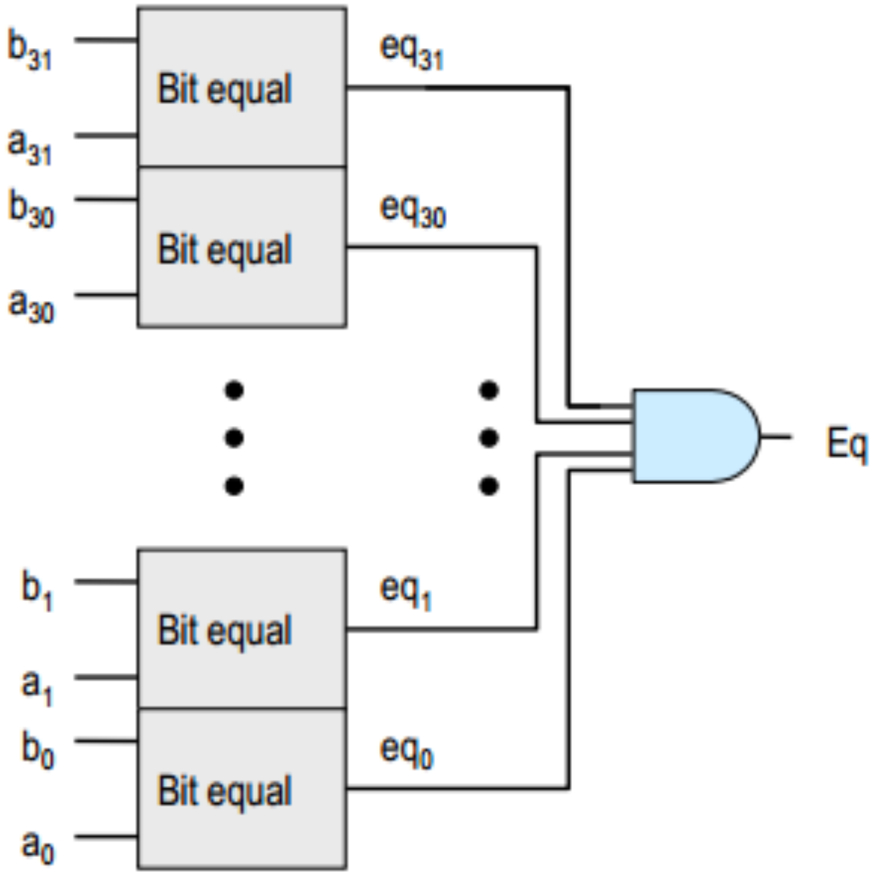
HCL Expression

```
bool eq = (a&&b)||(!a&&!b)
```

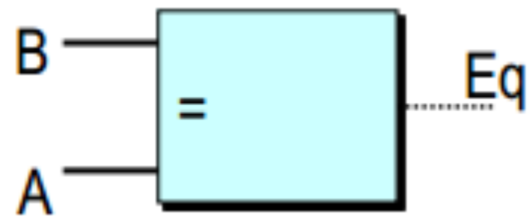
- Generate 1 if a and b are equal

- **Hardware Control Language (HCL)**
  - Very simple hardware description language
    - Boolean operations have syntax similar to C logical operations
  - We'll use it to describe control logic for processors

Word Equality



Word-Level Representation



HCL Representation

```
bool Eq = (A == B)
```

- **32-bit word size**
- **HCL representation**
  - Equality operation
  - Generates Boolean value

Thank you!

严思明
dantes@pku.edu.cn