

CS 33

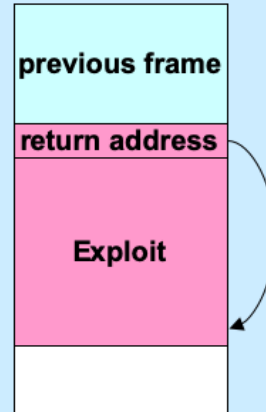
Machine Programming (6)

Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.

Buffer Overflow Attack Revisited

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

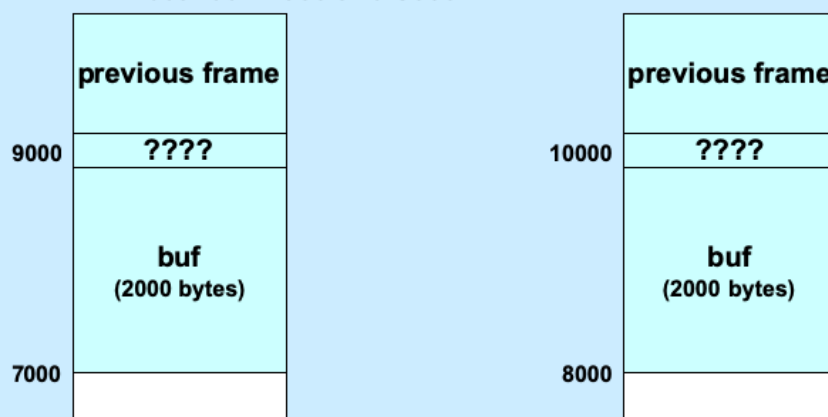
```
main:  
    subq $88, %rsp # grow stack  
    movq %rsp, %rdi # setup arg  
    call gets  
    movq %rsp, %rdi # setup arg  
    call puts  
    movl $0, %eax # set return value  
    addq $88, %rsp # pop stack  
    ret
```



We revisit our slide illustrating the buffer overflow attack. What's key is the attacker knows the address of the buffer, so they can overwrite the return address with the buffer address (after filling the buffer with the exploit code).

Stack Randomization

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



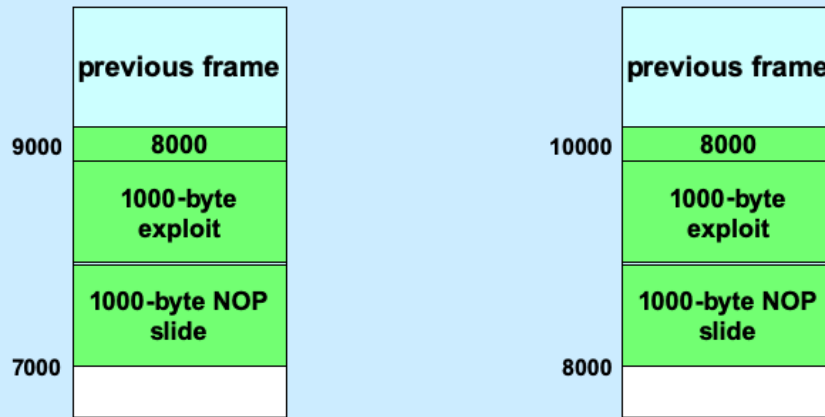
As mentioned, one way to make such attacks more difficult is to randomize the location of the buffer. Suppose it's not known exactly where the buffer begins, but it is known that it begins somewhere between 7000 and 8000. Thus it's not clear with what value to overwrite the return address of the stack frame being attacked.

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

A NOP slide is a sequence of NOP (no-op) instructions. Each such instruction does nothing, but simply causes control to move to the next instruction.

NOP Slides and Stack Randomization



To deal with stack randomization, we might simply pad the beginning of the exploit with a NOP slide. Thus, in our example, let's assume the exploit code requires 1000 bytes, and we have 1000 bytes of uncertainty as to where the stack ends (and the buffer begins). The attacker inputs 2000 bytes: the first 1000 are a NOP slide, the second 1000 are the actual exploit. The return address is overwritten with the highest possible buffer address (8000). If the buffer actually starts at its lowest possible address (7000), the return address points to the beginning of the actual exploit, which is executed immediately after the return takes place. But if the buffer starts at its highest possible address (8000), the return address points to the beginning of the NOP slide. Thus when the return takes place, control goes to the NOP slide, but soon gets to the exploit code.

Recursive Function

```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

- **Registers**

- **%eax, %edx** used without first saving
- **%ebx** used, but saved at beginning & restored at end

```
pcount_r:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    subl $4, %esp
    movl 8(%ebp), %ebx
    movl $0, %eax
    testl %ebx, %ebx
    je .L3
    movl %ebx, %eax
    shrl $1, %eax
    movl %eax, (%esp)
    call pcount_r
    movl %ebx, %edx
    andl $1, %edx
    leal (%edx, %eax), %eax
.L3:
    addl $4, %esp
    popl %ebx
    popl %ebp
    ret
```

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);  
}
```

The slide shows two implementations of the factorial function. Both use recursion. In the version on the left, the result of each recursive call is used within the invocation that issued the call. In the second, the result of each recursive call is simply returned. This is known as *tail recursion*.

No Tail Recursion (1)

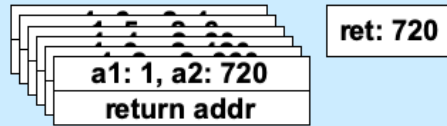
| |
|-------------|
| x: 6 |
| return addr |
| x: 5 |
| return addr |
| x: 4 |
| return addr |
| x: 3 |
| return addr |
| x: 2 |
| return addr |
| x: 1 |
| return addr |

Here we look at the stack usage for the version without tail recursion. Note that we have as many stack frames as the value of the argument; the results of the calls are combined after the stack reaches its maximum size.

No Tail Recursion (2)

| | |
|-------------|----------|
| x: 6 | ret: 720 |
| return addr | |
| x: 5 | ret: 120 |
| return addr | |
| x: 4 | ret: 24 |
| return addr | |
| x: 3 | ret: 6 |
| return addr | |
| x: 2 | ret: 2 |
| return addr | |
| x: 1 | ret: 1 |
| return addr | |

Tail Recursion



With tail recursion, since the result of the recursive call is not used by the issuing stack frame, it's possible to reuse the issuing stack frame to handle the recursive invocation. Thus rather than push a new stack frame on the stack, the current one is written over. Thus the entire sequence of recursive calls can be handled within a single stack frame.

Code: gcc -O1

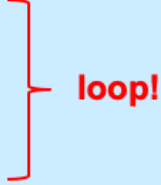
```
f2:
    movl    %esi, %eax
    cmpl    $1, %edi
    je      .L5
    subq    $8, %rsp
    movl    %edi, %esi
    imull   %eax, %esi
    subl    $1, %edi
    call    f2      # recursive call!
    addq    $8, %rsp
.L5:
    rep
    ret
```

This is the result of compiling the tail-recursive version of factorial using gcc with the -O1 flag. This flag turns on a moderate level of code optimization, but not enough to cause the stack frame to be reused.

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
```



Here we've compiled the program using the `-O2` flag, which turns on additional optimization (at the cost of increased compile time), with the result that the recursive calls are optimized away — they are replaced with a loop.

Why not always compile with `-O2`? For “production code” that is bug-free (assuming this is possible), this is a good idea. But this and other aggressive optimizations make it difficult to relate the runtime code with the source code. Thus, a runtime error might occur at some point in the program's execution, but it is impossible to determine exactly which line of the source code was in play when the error occurred.

Quiz 1

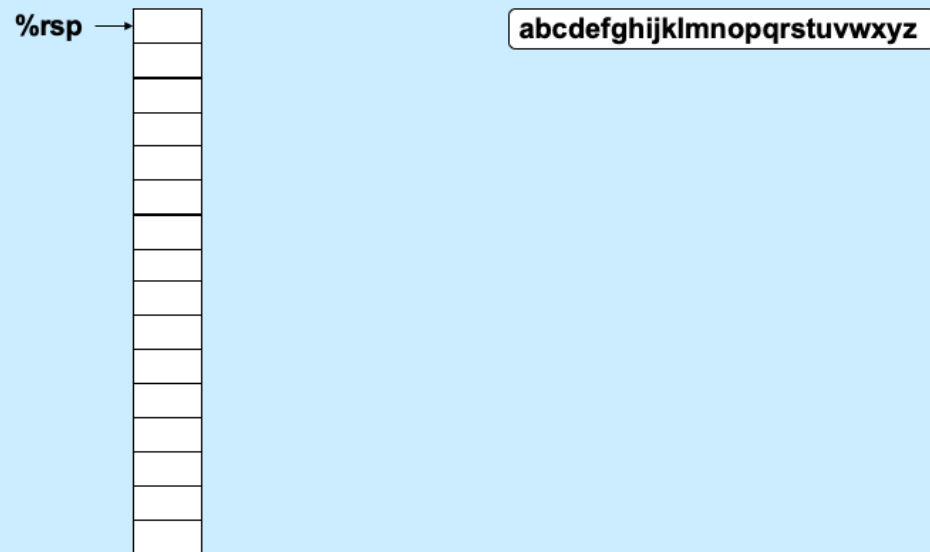
```
int main() {  
    recur();  
    return 0;  
}  
  
void recur() {  
    char c = getchar();  
    if (c != EOF) {  
        recur();  
        putchar(c);  
    }  
}
```

• What does this program do?

- a) repeatedly: reads a char, then writes it
- b) reads in all its input, then writes it out in the order it was read in
- c) reads in all its input, then writes it all out in reverse order
- d) reads in all of its input backwards, then writes it all out

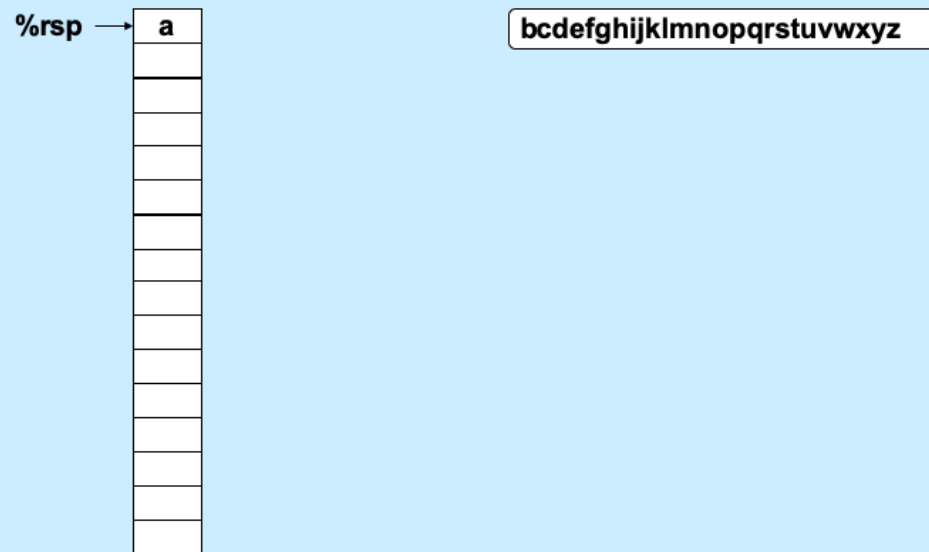
The function *getchar* reads (and returns) the next input character. The function *putchar* outputs its argument.

Reversing the Input



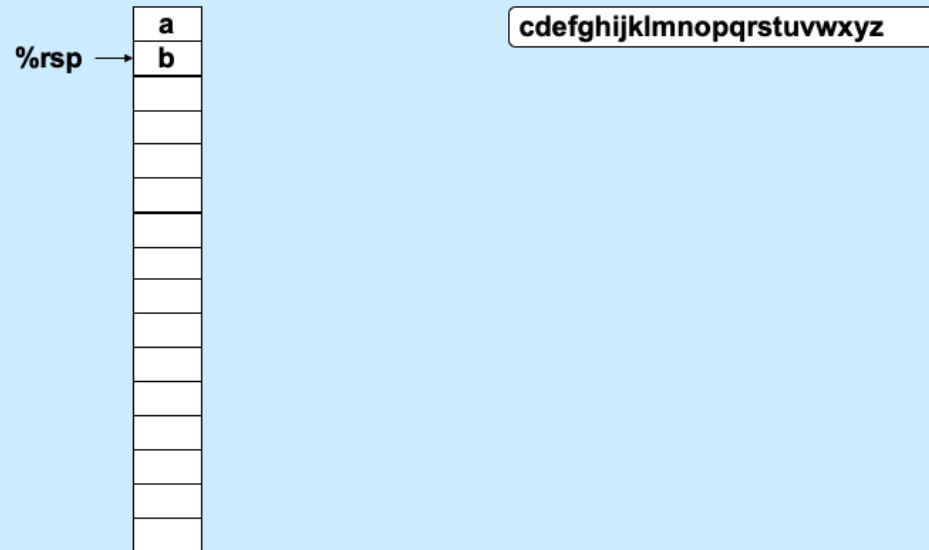
Suppose we could store the characters as we read them in into contiguous locations on the stack. In the slide, the stack is set up for this and we're about to starting reading the input, consisting of the letters of the alphabet.

Reversing the Input



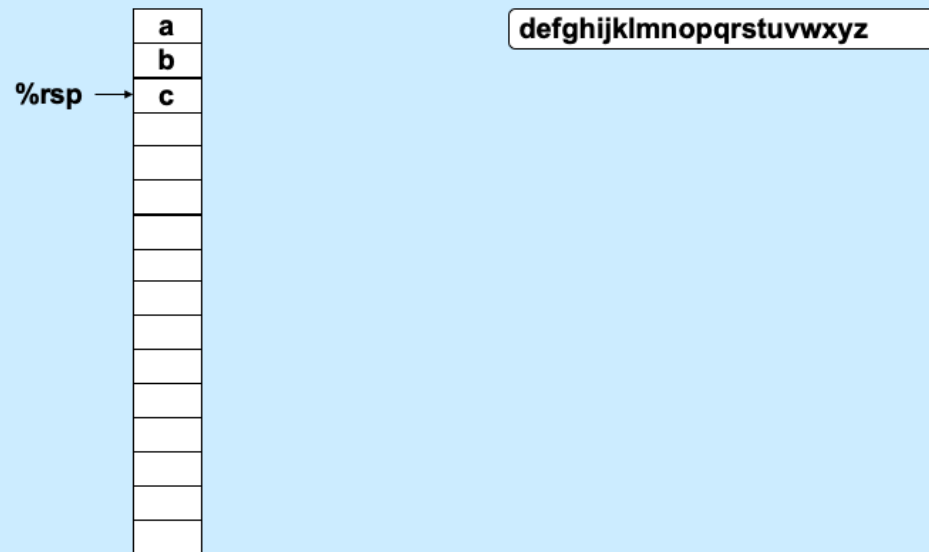
We read the first input character and it's pushed on the stack.

Reversing the Input



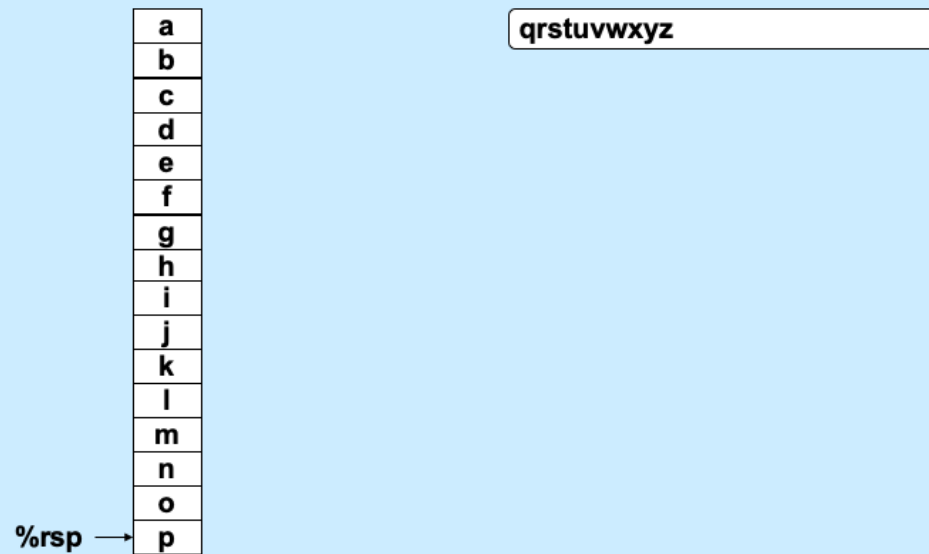
We read the second input character and it's pushed on the stack.

Reversing the Input



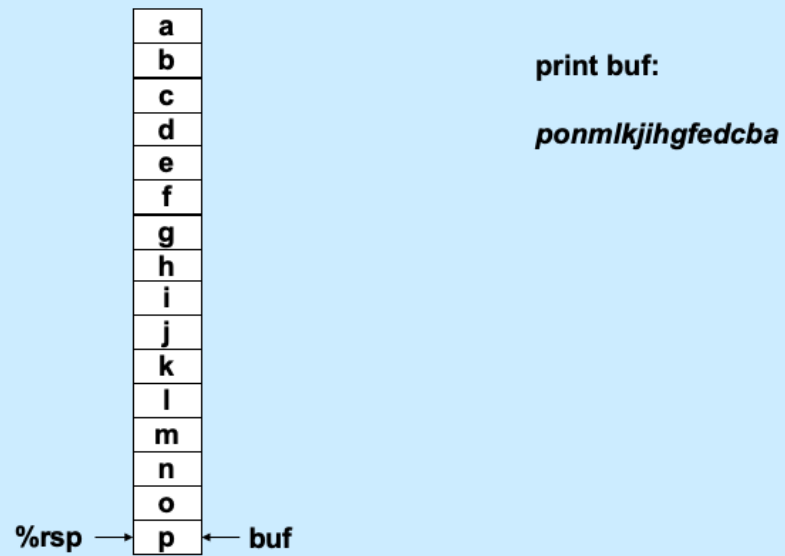
Third is pushed on the stack ...

Reversing the Input



..., as are the rest (up through p).

Reversing the Input



print buf:

ponmlkjihgfedcba

If `buf` points to the current end of the stack and we print the stack starting with `p`, we get what's shown.

Doing it in C (Sort of)

```
int main( ) {
    char *buf;
    unsigned long cnt=0;
    long i;
    unsigned long ssize;

    for (ssize=16; ; ssize += 16) {
        Alloc16BytesOnStack(buf);
        // macro that modifies buf
        // to point to next 16 bytes
        for (i=15; i>=0; i--, cnt++) {
            if ((buf[i] =
                getchar()) == EOF)
                goto done;
        }
    }

done:
    write(1, &buf[i+1], cnt);
    write(1, "\n", 1);

    PopBytesOffStack(ssize);
    return 0;
}
```

We'd like to express in C what we did in the past few slides. This can't be done directly since there isn't an easy way to allocate arbitrary storage on the stack. (There is a function *alloca* that sort of does this, but its not defined particularly well and its implementation differs on Linux and OS X.) So we hypothesize (and then implement) the needed functionality.

Alloc16BytesOnStack is implemented as a macro (and thus its argument can be modified). It grows the stack by 16 bytes and sets its argument to the new end of the stack. Thus there are 16 bytes of additional stack space into which we read the next 16 bytes of input. After reading all the input, the system-call *write* is invoked (it's the lowest-level output function and interacts directly with the operating system). Its first argument (1) says to send the output to *stdout*, which is normally the window in which the program is running. The second argument is the address of the beginning of the data that is to be written. Its last argument is how many bytes to write. Thus the first call to *write* outputs all the characters that have been read in, but in reverse order. The second call to *write* simply outputs a newline character.

PopBytesOffStack adds to the stack pointer register the value of its argument. In this case, it restores the stack pointer to what it was before the outermost for loop began execution.

We show in the source code associated with this lecture how *Alloc16BytesOnStack* and *PopBytesOffStack* are implemented (it involves embedding x86 assembler code into the object code produced by gcc). You're not responsible for knowing how this is done – it's beyond the scope of the course.

Computer Architecture and Optimization (1)

What You Need to Know to Write Better Code

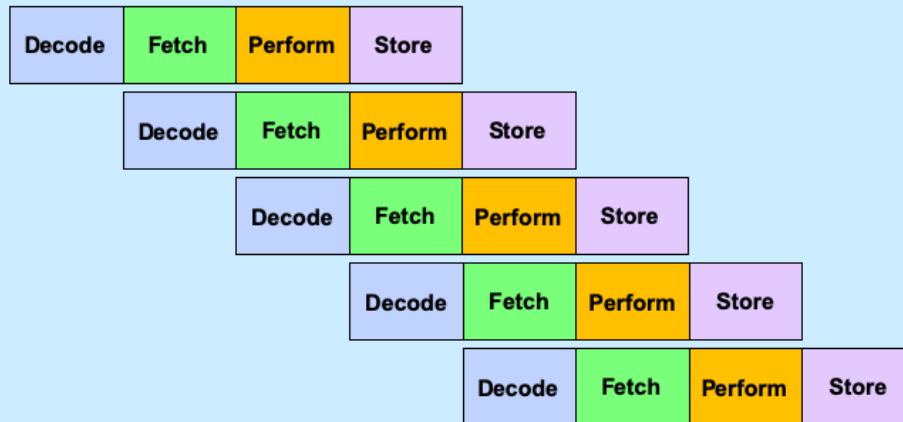
Simplistic View of Processor

```
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);  
}
```

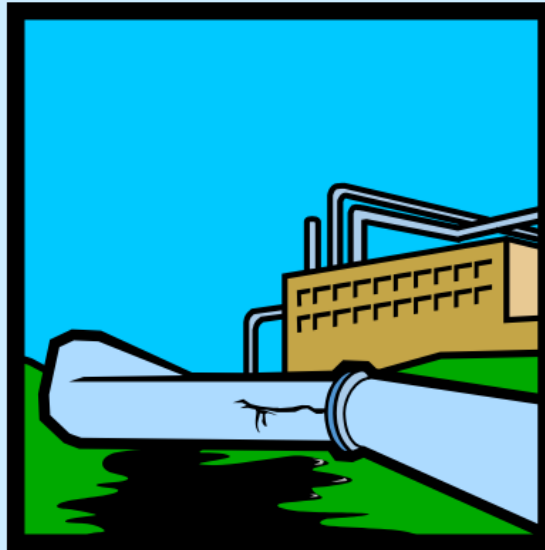
Pipelines



Analysis

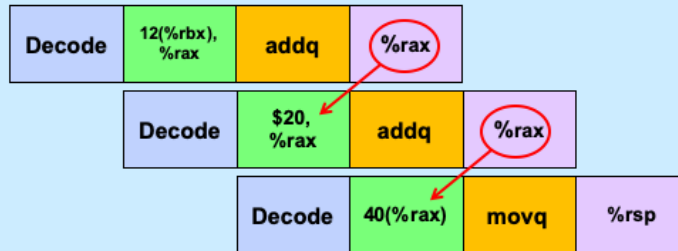
- **Not pipelined**
 - each instruction takes, say, 3.2 nanoseconds
 - » 3.2 ns latency
 - 312.5 million instructions/second (MIPS)
- **Pipelined**
 - each instruction still takes 3.2 ns
 - » latency still 3.2 ns
 - an instruction completes every .8 ns
 - » 1.25 billion instructions/second (GIPS) throughput

Hazards ...

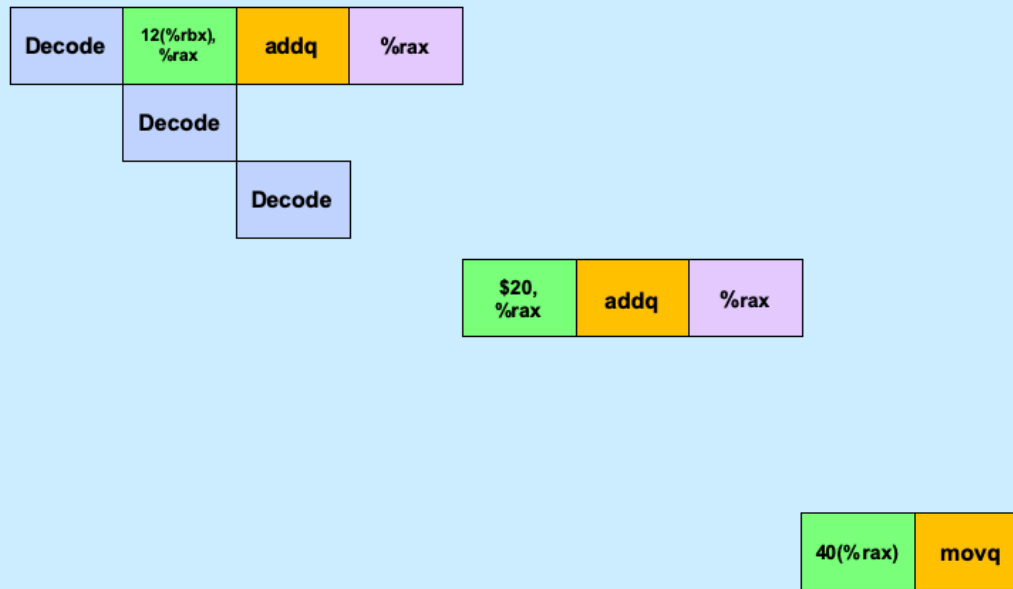


Data Hazards

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



Coping

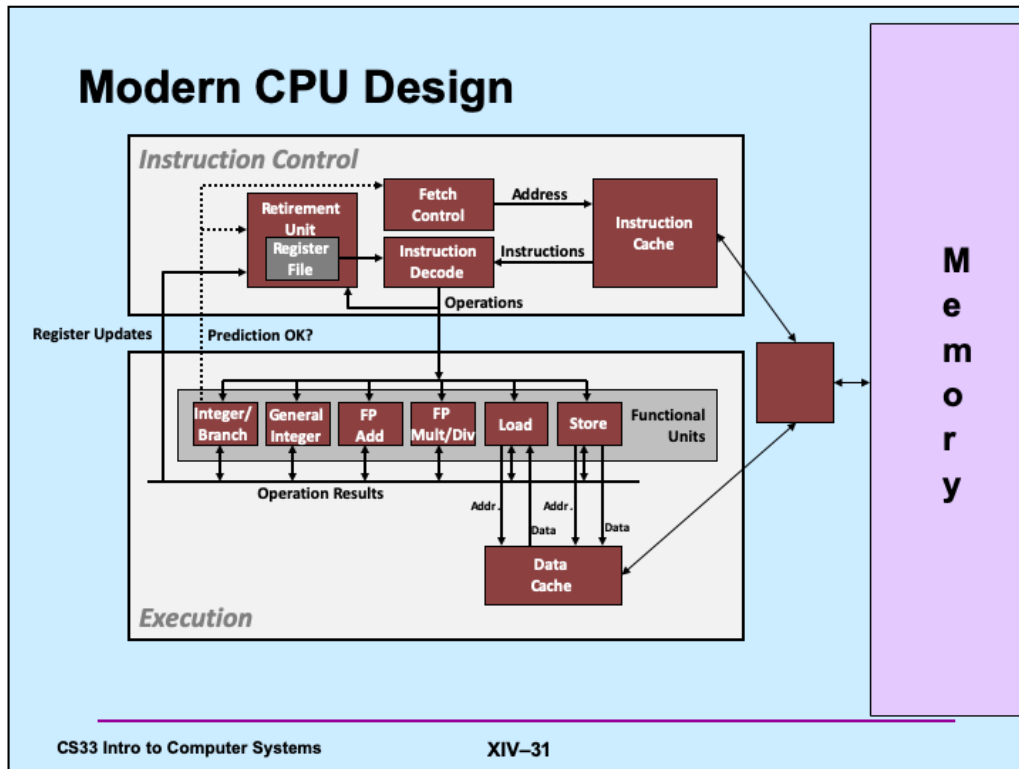


Control Hazards

```
    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



Adapted from slide supplied by CMU.

Note that the functional units operate independently of one another. Thus, for example, the floating-point add unit can be working on one instruction, which the general integer unit can be working on another. Thus there are additional possibilities for parallel execution of instructions.

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

Supplied by CMU.

Optimizing Compilers

- **Provide efficient mapping of program to machine**
 - register allocation
 - code selection and ordering (scheduling)
 - dead code elimination
 - 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

Supplied by CMU.

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**

Supplied by CMU.

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];
```

Compiler-Generated Code Motion

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

```
set_row:
    testq    %rcx, %rcx          # Test n
    jle      .L1                 # If 0, goto done
    imulq    %rcx, %rdx          # i *= n
    leaq     (%rdi,%rdx,8), %rdi  # rowp = A + n*i*8
    movl     $0, %eax            # j = 0
.L3:                                     # loop:
    movq     (%rsi,%rax,8), %rdx  # t = b[j]
    movq     %rdx, (%rdi,%rax,8)  # rowp[j] = t
    addq     $1, %rax             # j++
    cmpq     %rcx, %rax          # Compare n:j
    jg       .L3                 # If >, goto loop
.L1:                                     # done:
    rep ; ret
```

Supplied by CMU, updated for current gcc. This happens when gcc is given the -O1 flag, specifying that it perform a small amount of code optimization.

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
 - $16 * x \quad \rightarrow \quad x \ll 4$
 - utility is machine-dependent
 - depends on cost of multiply or divide instruction
 - » on Intel Nehalem, integer multiply requires 3 CPU cycles
- 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;  
}
```

Supplied by CMU.

Note that an integer addition requires one CPU cycle. gcc can recognize optimizations of the sort shown here.

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;
```

3 multiplications: $i*n$, $(i-1)*n$, $(i+1)*n$

```
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
```

```
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;
```

1 multiplication: $i*n$

```
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
```

Supplied by CMU.

The code in the lower-left box is what gcc produced for the code in the upper left box. On the right is a much better version that was done by hand.

Quiz 2

The fastest means for evaluating

$$n*n + 2*n + 1$$

requires exactly:

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

Optimization Blocker #1: Function Calls

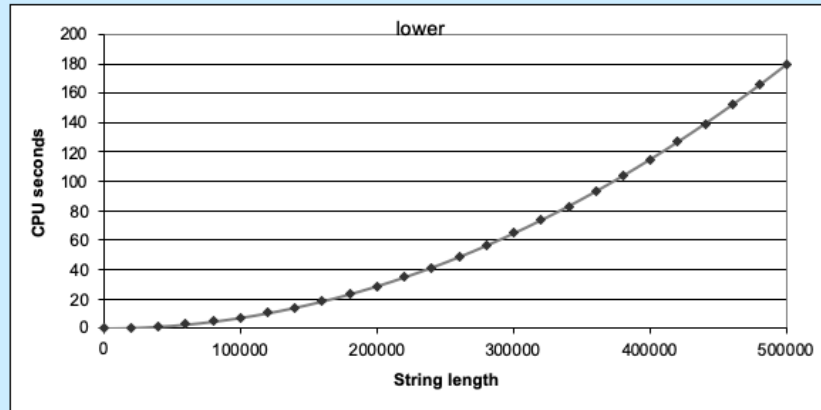
- Function to convert string to lower case

```
void lower(char *s){  
    int i;  
    for (i = 0; i < strlen(s); i++)  
        if (s[i] >= 'A' && s[i] <= 'Z')  
            s[i] -= ('A' - 'a');  
}
```

Supplied by CMU.

Lower Case Conversion Performance

- Time quadruples when string length doubles
- Quadratic performance



Supplied by CMU.

Convert Loop To Goto Form

```
void lower(char *s){
    int i = 0;
    if (i >= strlen(s))
        goto done;
loop:
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
    i++;
    if (i < strlen(s))
        goto loop;
done:
}
```

- **strlen** executed every iteration

Calling Strlen

```
size_t strlen(const char *s){
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

- **strlen performance**
 - only way to determine length of string is to scan its entire length, looking for null character
- **Overall performance, string of length N**
 - N calls to strlen
 - overall $O(N^2)$ performance

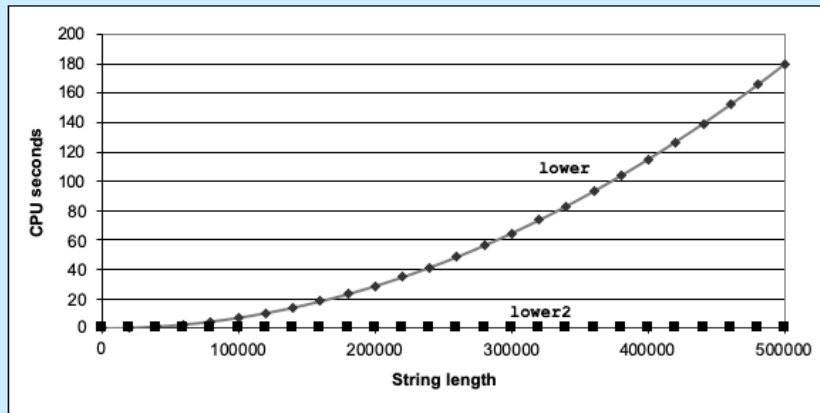
Improving Performance

```
void lower2(char *s) {  
    int i;  
    int len = strlen(s);  
    for (i = 0; i < len; i++)  
        if (s[i] >= 'A' && s[i] <= 'Z')  
            s[i] -= ('A' - 'a');  
}
```

- **Move call to `strlen` outside of loop**
 - since result does not change from one iteration to another
 - form of code motion

Lower-Case Conversion Performance

- Time doubles when string-length doubles
 - linear performance of lower2



Supplied by CMU.

Optimization Blocker: Function Calls

- *Why couldn't compiler move strlen out of inner loop?*
 - function may have side effects
 - » alters global state each time called
 - function may not return same value for given arguments
 - » depends on other parts of global state
 - » function lower could interact with strlen
- **Warning:**
 - compiler treats procedure call as a black box
 - weak optimizations near them
- **Remedies:**
 - use of inline functions
 - » gcc does this with -O2
 - do your own code motion

```
int lencnt = 0;
size_t strlen(const char *s) {
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```

Memory Matters

```
/* Sum rows of n X n matrix a
   and store result in vector b */
void sum_rows1(long n, long a[][n], long *b) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i][j];
    }
}
```

```
# sum_rows1 inner loop
.L3:
    movq    (%r8,%rax,8), %rcx    # rcx = a[i][j]
    addq    %rcx, (%rdx)          # b[i] += rcx
    addq    $1, %rax              # j++
    cmpq    %rax, %rdi            # if i<n
    jne     .L3                   # goto .L3
```

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

Based on a slide supplied by CMU.

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store result in vector b */
void sum_rows1(long n, long a[][n], long *b) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i][j];
    }
}
```

```
int A[3][3] =
    {{ 0, 1, 2},
     { 4, 8, 16},
     {32, 64, 128}};

int *B = &A[1][0];

sum_rows1(3, A, B;
```

Value of B:

init: [4, 8, 16]

i = 0: [3, 8, 16]

i = 1: [3, 22, 16]

i = 2: [3, 22, 224]

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

Supplied by CMU, updated for current gcc.

Removing Aliasing

```
/* Sum rows of n X n matrix a
   and store result in vector b */
void sum_rows1(long n, long a[][n], long *b) {
    long i, j;
    for (i = 0; i < n; i++) {
        long val = 0;
        for (j = 0; j < n; j++)
            val += a[i][j];
        b[i] = val;
    }
}
```

```
# sum_rows2 inner loop
.L4:
    addq    (%r8, %rax, 8), %rcx
    addq    $1, %rdi
    cmpq    %rcx, %rdi
    jne     .L4
```

- No need to store intermediate results

Optimization Blocker: Memory Aliasing

- **Aliasing**

- two different memory references specify single location
- easy to have happen in C
 - » since allowed to do address arithmetic
 - » direct access to storage structures
- get in habit of introducing local variables
 - » accumulating within loops
 - » **your way of telling compiler not to check for aliasing**

Supplied by CMU.

C99 to the Rescue

- **New attribute**

- **restrict**

- » applied to a pointer, tells the compiler that the object pointed to will be accessed only via this pointer
 - » compiler thus doesn't have to worry about aliasing
 - » but the programmer does ...
 - » **syntax**

```
int *restrict pointer;
```

Pointers and Arrays

- `long a[][n]`
 - **a is a 2-D array of longs, the size of each row is n**
- `long (*b)[n]`
 - **b is a pointer to a 1-D array of size n**
- **a and b are of the same type**

Memory Matters, Fixed

```
/* Sum rows of n X n matrix a
   and store result in vector b */
void sum_rows1(long n, long (*restrict a)[n], long *restrict b) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i][j];
    }
}
```

```
# sum_rows1 inner loop
.L3:
    addq    (%rdi), %rax
    addq    $8, %rdi
    cmpq    %rcx, %rdi
    jne     .L3
```

- Code doesn't update `b[i]` on every iteration

Note: we must give gcc the flag “-std=gnu99” for this to be compiled.

Observe that

```
long (*a)[n]
```

declares `a` to be a pointer to an array of `n` longs.

Thus

```
long (*restrict a)[n]
```

declares `a` to be a restricted pointer to an array of `n` longs

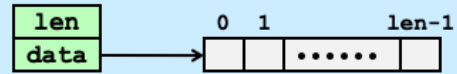
Exploiting Instruction-Level Parallelism

- **Need general understanding of modern processor design**
 - hardware can execute multiple instructions in parallel
- **Performance limited by data dependencies**
- **Simple transformations can have dramatic performance improvement**
 - compilers often cannot make these transformations
 - lack of associativity and distributivity in floating-point arithmetic

Supplied by CMU.

Benchmark Example: Datatype for Vectors

```
/* data structure for vectors */  
typedef struct{  
    int len;  
    data_t *data;  
} vec_t, *vec_ptr_t;
```



```
/* retrieve vector element and store at val */  
int get_vec_element(vec_ptr_t v, int idx, data_t *val){  
    if (idx < 0 || idx >= v->len)  
        return 0;  
    *val = v->data[idx];  
    return 1;  
}  
  
/* return length of vector */  
int vec_length(vec_ptr_t v) {  
    return v->len;  
}
```

Benchmark Computation

```
void combinel(vec_ptr_t v, data_t *dest){
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or
product of vector
elements

- **Data Types**

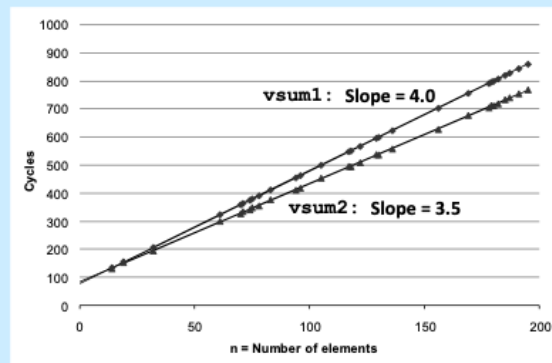
- use different declarations
for **data_t**
 - » **int**
 - » **float**
 - » **double**

- **Operations**

- use different definitions of
OP and IDENT
 - » **+**, **0**
 - » *****, **1**

Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- $T = \text{CPE} \cdot n + \text{Overhead}$
 - CPE is slope of line



Supplied by CMU.

Benchmark Performance

```
void combinel(vec_ptr_t v, data_t *dest){
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or
product of vector
elements

| Method | Integer | | Double FP | |
|----------------------|---------|------|-----------|------|
| Operation | Add | Mult | Add | Mult |
| Combine1 unoptimized | 29.0 | 29.2 | 27.4 | 27.9 |
| Combine1 -O1 | 12.0 | 12.0 | 12.0 | 13.0 |

Supplied by CMU.

Move vec_length

```
void combine2(vec_ptr_t v, data_t *dest){
    long int i;
    long int length = vec_length(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

| Method | Integer | | Double FP | |
|----------------------|---------|------|-----------|-------|
| Operation | Add | Mult | Add | Mult |
| Combine1 unoptimized | 29.0 | 29.2 | 27.4 | 27.9 |
| Combine1 -O1 | 12.0 | 12.0 | 12.0 | 13.0 |
| Combine2 | 8.03 | 8.09 | 10.09 | 12.08 |

Supplied by CMU.

Eliminate Function Calls

```
void combine3(vec_ptr_t v, data_t *dest){
    long int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

```
data_t *get_vec_start(
    vec_ptr v) {
    return v->data;
}
```

| Method | Integer | | Double FP | |
|-----------|---------|------|-----------|-------|
| Operation | Add | Mult | Add | Mult |
| Combine2 | 8.03 | 8.09 | 10.09 | 12.08 |
| Combine3 | 6.01 | 8.01 | 10.01 | 12.02 |

Eliminate Unneeded Memory References

```
void combine4(vec_ptr_t v, data_t *dest){
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

| Method | Integer | | Double FP | |
|--------------|---------|------|-----------|------|
| Operation | Add | Mult | Add | Mult |
| Combine1 -O1 | 12.0 | 12.0 | 12.0 | 13.0 |
| Combine4 | 2.0 | 3.0 | 3.0 | 5.0 |

Supplied by CMU.

Quiz 3

Combine4 is pretty fast; we've done all the "obvious" optimizations. How much faster will we be able to make it? (Hint: it involves taking advantage of pipelining and multiple functional units on the chip.)

- a) 1× (it's already as fast as possible)**
- b) 2× – 4×**
- c) 16× – 64×**