Code Optimization and Linking

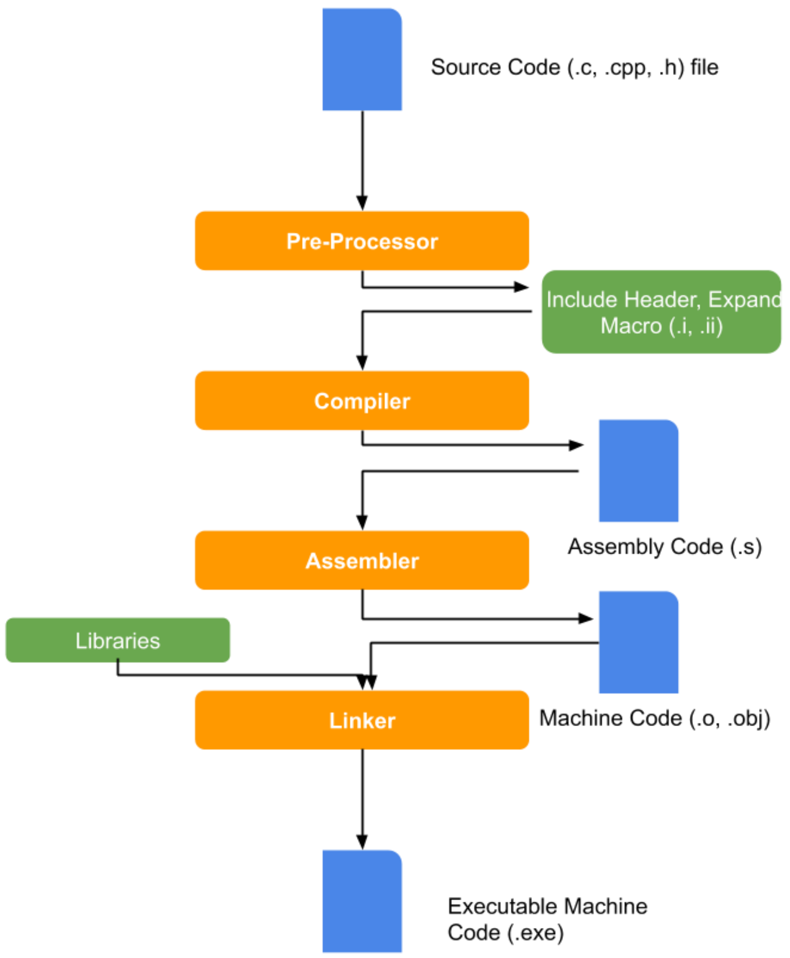
15-213/18-213/15-513: Introduction to Computer Systems
12th Lecture, October 7, 2021

Today

- **Basics of compiler optimization**
  - Principles and goals
  - Some example optimizations
  - Obstacles to optimization
- **Linking: combining object files into programs**
  - Symbols and symbol resolution
  - Relocation
  - Static libraries
- **Quiz**
- **If we have time**
  - Branch prediction
  - Dynamic libraries

What does it mean to compile code?

- The CPU only understands *machine code* directly
- All other languages must be either
  - *interpreted:* executed by software
  - *compiled:* translated to machine code by software



There’s a story that starts like this:

Back in the Good Old Days,
when the term "software" sounded funny
and Real Computers were made out of drums
and vacuum tubes,
Real Programmers wrote in machine code.

Not FORTRAN. Not RATFOR. Not, even,
assembly language.

Machine Code.

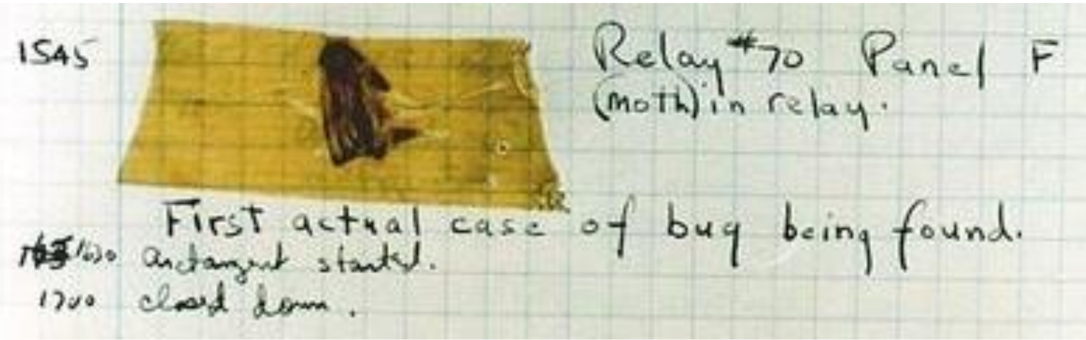
Raw, unadorned, inscrutable hexadecimal numbers. Directly.

— “The Story of Mel, a Real Programmer”

Ed Nather, 1983

- **Rear Admiral Grace Hopper**
  - **Invented first compiler in 1951 (technically it was a linker)**
  - **Coined “compiler” (and “bug”)**
  - **Compiled for Harvard Mark I**
  - **Eventually led to COBOL**

    (which ran the world for years)
  - **“I decided data processors ought to be able to write their programs in English, and the computers would translate them into machine code”**



- **John Backus**
  - **Led team at IBM invented the first commercially available compiler in 1957**
  - **Compiled FORTRAN code for the IBM 704 computer**
  - **FORTRAN still in use today for high performance code**
  - **"Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, I started work on a programming system to make it easier to write programs"**

- **Fran Allen**
  - **Pioneer of many optimizing compilation techniques**
  - **Wrote a paper simply called "Program Optimization" in 1966**
  - **"This paper introduced the use of graph-theoretic structures to encode program content in order to automatically and efficiently derive relationships and identify opportunities for optimization"**
  - **First woman to win the ACM Turing Award (the "Nobel Prize of Computer Science")**

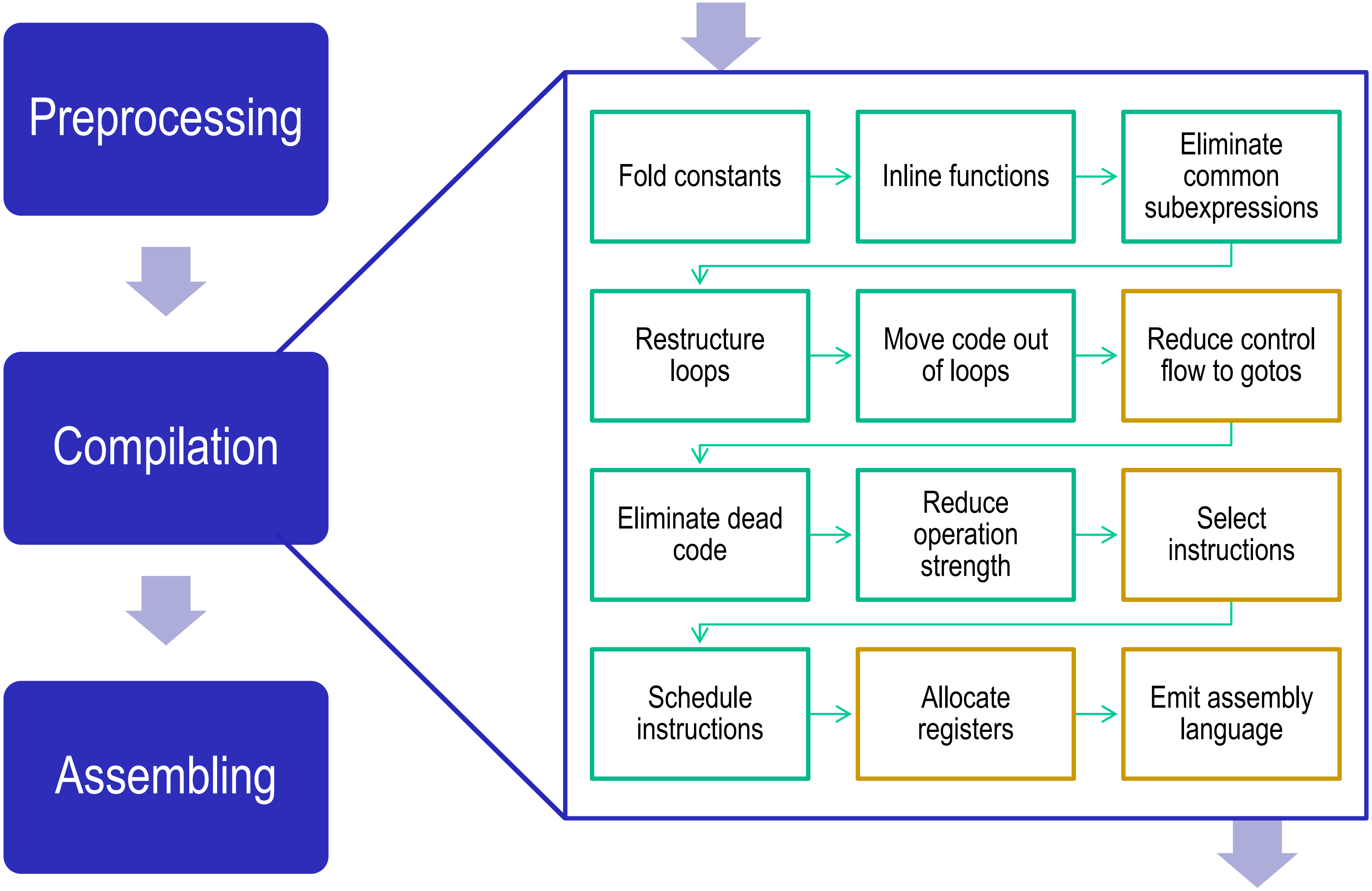
Goals of compiler optimization

- **Minimize number of instructions**
  - Don’t do calculations more than once
  - Don’t do unnecessary calculations at all
  - Avoid slow instructions (multiplication, division)
- **Avoid waiting for memory**
  - Keep everything in registers whenever possible
  - Access memory in cache-friendly patterns
  - Load data from memory early, and only once
- **Avoid branching**
  - Don’t make unnecessary decisions at all
  - Make it easier for the CPU to predict branch destinations
  - “Unroll” loops to spread cost of branches over more instructions

Limits to compiler optimization

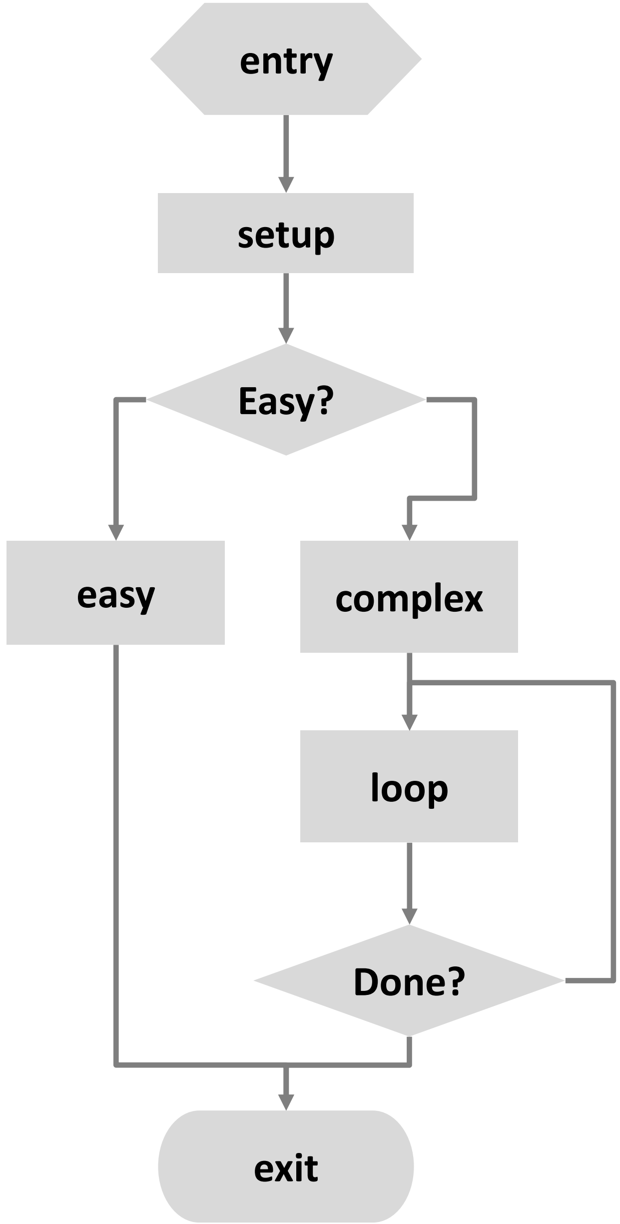
- **Generally cannot improve algorithmic complexity**
  - Only constant factors, but those can be worth 10x or more…
- **Must not cause *any* change in program behavior**
  - Programmer may not care about "edge case" behavior, but compiler does not know that
  - Exception: language may declare some changes acceptable
- **Usually only analyze one function at a time**
  - Whole-program analysis is usually too expensive
  - Exception: *inlining* merges many functions into one
- **Cannot anticipate run-time inputs**
  - "Worst case" performance can be just as important as "normal"
  - Especially for code exposed to *malicious* input (e.g. network servers)

Compilation is a pipeline



Two kinds of optimizations

- **Local optimizations work inside a single *basic block***
  - Constant folding, strength reduction, (local) CSE, …
- **Global optimizations process the entire *control flow graph* of a function**
  - Loop nest optimization, code motion, (global) CSE, dead code elimination, …



Constant Folding

- **Do arithmetic in the compiler**

```
long mask = 0xFF << 8;      →
long mask = 0xFF00;
```

- **Any expression with constant inputs can be folded**
- **Might even be able to remove library calls…**

```
size_t namelen = strlen("Harry Bovik");    →
size_t namelen = 11;
```

Strength reduction

- **Replace expensive operations with cheaper ones**

```
long a = b * 5;     →
long a = (b << 2) + b;
```

- **Multiplication and division are the usual targets**
- **Multiplication is often hiding in memory access expressions**

Dead code elimination

- **Don't emit code that will never be executed**

  ~~`if (0) { puts("Kilroy was here"); }`~~
  ~~`if (1) {`~~ `puts("Only bozos on this bus");` ~~`}`~~

- **Don't emit code whose result is overwritten**

  ~~`x = 0;`~~
  `x = 23;`

- **These may look silly, but...**
  - Can be produced by other optimizations
  - Assignments to x might be far apart

Common Subexpression Elimination

- **Factor out repeated calculations, only do them once**

```
norm[i] = v[i].x*v[i].x + v[i].y*v[i].y;
```

→

```
elt = &v[i];
x = elt->x;
y = elt->y;
norm[i] = x*x + y*y;
```

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
         + pred(0)
         + pred(y+1);
}
```

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
         + pred(0)
         + pred(y+1);
}
```

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

Always true **Does nothing** **Can constant fold**

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

```
int func(int y) {
  int tmp = 0;
  if (y != 0) tmp = y - 1;

  if (y != -1) tmp += y;
  return tmp;
}
```

Code Motion

- Move calculations out of a loop
- Only valid if every iteration would produce same result

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

→

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

Loop Unrolling

- Amortize cost of loop condition by duplicating body
- Creates opportunities for CSE, code motion, scheduling
- Prepares code for vectorization
- Can hurt performance by increasing code size

```
for (size_t i = 0; i < nelts; i++) {
    A[i] = B[i]*k + C[i];
}
```

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    A[i  ] = B[i  ]*k + C[i  ];
    A[i+1] = B[i+1]*k + C[i+1];
    A[i+2] = B[i+2]*k + C[i+2];
    A[i+3] = B[i+3]*k + C[i+3];
}
```

Loop Unrolling

- Amortize cost of loop condition by duplicating body
- Creates opportunities for CSE, code motion, scheduling
- Prepares code for vectorization
- Can hurt performance by increasing code size

```
for (size_t i = 0; i < nelts; i++) {
    A[i] = B[i]*k + C[i];
}
```

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    A[i  ] = B[i  ]*k + C[i  ];
    A[i+1] = B[i+1]*k + C[i+1];
    A[i+2] = B[i+2]*k + C[i+2];
    A[i+3] = B[i+3]*k + C[i+3];
}
```

When would this change be incorrect?

Scheduling

- Find the CPU something useful to do while it's waiting for memory, division unit, etc.
- Extremely machine-dependent, but here's a basic example:

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    A[i  ] = B[i  ]*k + C[i  ];
    A[i+1] = B[i+1]*k + C[i+1];
    A[i+2] = B[i+2]*k + C[i+2];
    A[i+3] = B[i+3]*k + C[i+3];
}
```

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    B0 = B[i]; B1 = B[i+1]; B2 = B[i+2]; B3 = B[i+3];
    C0 = C[i]; C1 = C[i+1]; C2 = C[i+2]; C3 = B[i+3];
    A[i  ] = B0*k + C0;
    A[i+1] = B1*k + C1;
    A[i+2] = B2*k + C2;
    A[i+3] = B3*k + C3;
}
```

Scheduling

- Find the CPU something useful to do while it's waiting for memory, division unit, etc.
- Extremely machine-dependent, but here's a basic example:

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    A[i  ] = B[i  ]*k + C[i  ];
    A[i+1] = B[i+1]*k + C[i+1];
    A[i+2] = B[i+2]*k + C[i+2];
    A[i+3] = B[i+3]*k + C[i+3];
}
```

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    B0 = B[i]; B1 = B[i+1]; B2 = B[i+2]; B3 = B[i+3];
    C0 = C[i]; C1 = C[i+1]; C2 = C[i+2]; C3 = B[i+3];
    A[i  ] = B0*k + C0;
    A[i+1] = B1*k + C1;
    A[i+2] = B2*k + C2;
    A[i+3] = B3*k + C3;
}
```

When would *this* change be incorrect?

Memory Aliasing

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

```
# sum_rows1 inner loop
.L4:
        movsd    (%rsi,%rax,8), %xmm0      # FP load
        addsd    (%rdi), %xmm0             # FP add
        movsd    %xmm0, (%rsi,%rax,8)      # FP store
        addq     $8, %rdi
        cmpq     %rcx, %rdi
        jne      .L4
```

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

Memory Aliasing

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

```
double A[9] =
  { 0,   1,   2,
    4,   8,   16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

```
double A[9] =
  { 0,   1,   2,
    3,  22, 224},
   32,  64, 128};
```

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

Removing Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

```
# sum_rows2 inner loop
.L10:
        addsd   (%rdi), %xmm0       # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

- Use a local variable for intermediate results

Removing Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows3(double *restrict a, double *restrict b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
# sum_rows3 inner loop
.L12:
        addsd   (%rdi), %xmm0      # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L12
```

- Use `restrict` qualifier to tell compiler that a and b cannot alias
- Less reliable than using local variables

Removing Aliasing

```
subroutine sum_rows4(a, b, n)
    implicit none
    integer, parameter :: dp = kind(1.d0)
    real(kind=dp), dimension(:), intent(in) :: a
    real(kind=dp), dimension(:), intent(out) :: b
    integer, intent(in) :: n
    integer :: i, j
    do i = 1,n
        b(i) = 0
        do j = 1,n
            b(i) = b(i) + a(i*n + j)
        end
    end
end
```

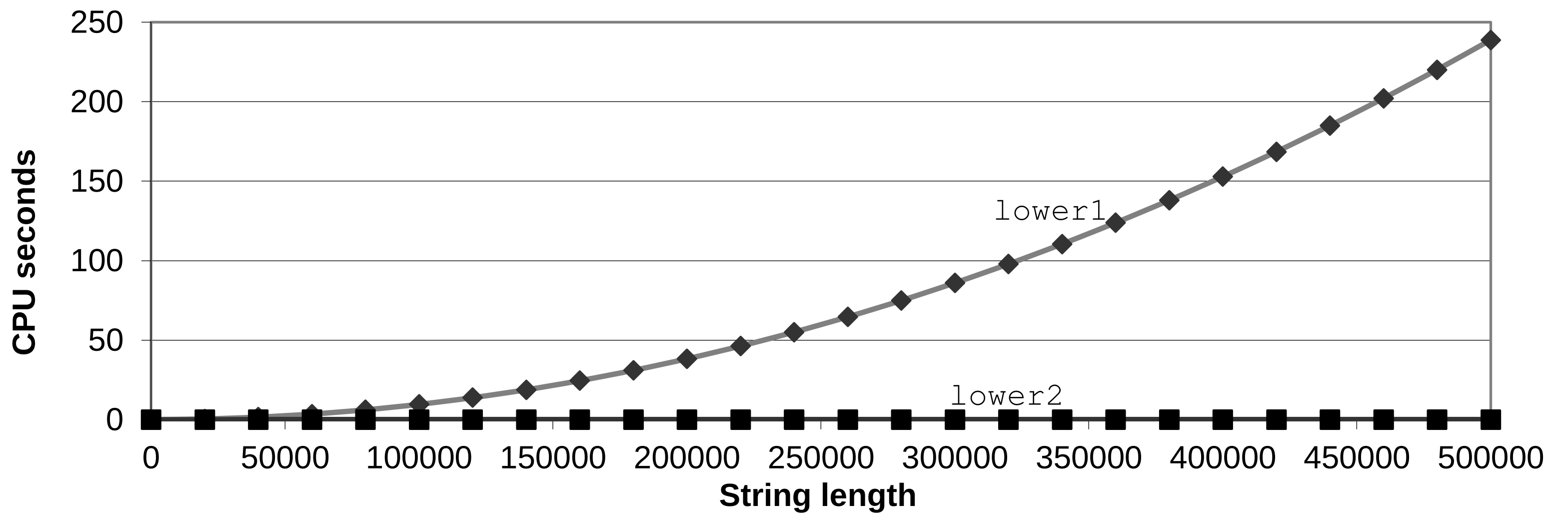
```
# sum_rows4 inner loop
.L5:
        addsd   (%rdi), %xmm0       # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L5
```

- Use Fortran
- Array parameters in Fortran are assumed not to alias

When the compiler can't move something

```
void lower1(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

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



Today

- Basics of compiler optimization
  - Principles and goals
  - Some example optimizations
  - Obstacles to optimization
- **Linking: combining object files into programs**
  - Symbols and symbol resolution
  - Relocation
  - Static libraries
- Quiz
- If we have time
  - Branch prediction
  - Dynamic libraries

Example C Program

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

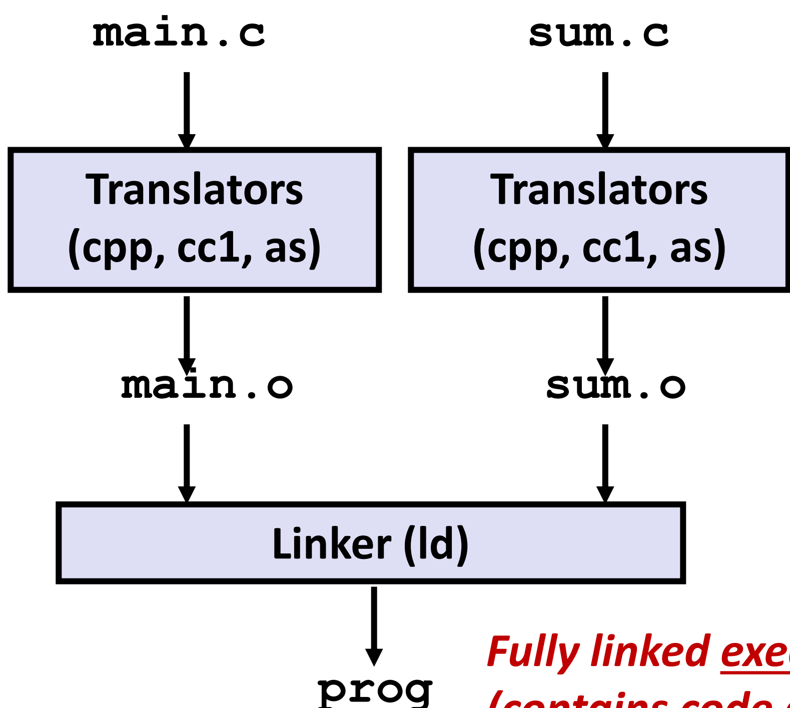
main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Linking

- Programs are translated and linked using a *compiler driver*:
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`



Source files

Separately compiled <u>relocatable</u> object files

Fully linked <u>executable</u> object file (contains code and data for all functions defined in `main.c` and `sum.c`)

What Do Linkers Do?

- **Step 1: Symbol resolution**

  - Programs define and reference *symbols* (global variables and functions):
    - `void swap() {…}     /* define symbol swap */`
    - `swap();             /* reference symbol swap */`
    - `int *xp = &x;       /* define symbol xp, reference x */`

  - Symbol definitions are stored in object file (by assembler) in *symbol table*.
    - Symbol table is an array of entries
    - Each entry includes name, size, and location of symbol.

  - **During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.**

Symbols in Example C Program

Definitions

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Reference

Linker Symbols

- **Every object file *m* has a table of symbols it defines or needs.**
- **Three types:**
- **Global definitions**
  - Symbols defined by *m* that can be referenced by other files.
  - In C, non-`static` functions and global variables.

- **Local definitions**
  - Symbols that are defined by *m* but *cannot* be referenced by other files.
  - In C, functions and global variables defined with `static`.
  - **Local linker symbols are *not* local program variables**

- **External references**
  - Symbols that *m* uses but does not define.
  - These must be defined by some other module.

Symbol Resolution

???

```
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Relocation Entries

```c
int array[2] = {1, 2};

int main(int argc, char**
argv)
{
    int val = sum(array, 2);
    return val;
}
```
main.c

```
0000000000000000 <main>:
   0:   48 83 ec 08           sub    $0x8,%rsp
   4:   be 02 00 00 00        mov    $0x2,%esi
   9:   bf 00 00 00 00        mov    $0x0,%edi       # %edi = &array
                      a: R_X86_64_32 array           # Relocation entry

   e:   e8 00 00 00 00        callq  13 <main+0x13>  # sum()
                      f: R_X86_64_PC32 sum-0x4       # Relocation entry
  13:   48 83 c4 08           add    $0x8,%rsp
  17:   c3                    retq
```
main.o

Source: `objdump -r -d main.o`

Symbol Identification

Which **of the following names will be in the symbol table of `symbols.o`?**

symbols.c:

```
int incr = 1;
static int foo(int a) {
  int b = a + incr;
  return b;
}

int main(int argc,
         char* argv[]) {
  printf("%d\n", foo(5));
  return 0;
}
```

Names:

- `incr`
- `foo`
- `a`
- `argc`
- `argv`
- `b`
- `main`
- `printf`
- `"%d\n"`

Can find this with `readelf`:

```
linux> readelf -s symbols.o
```

Local Symbols

- **Local non-static C variables vs. local static C variables**
  - Local non-static C variables: stored on the stack
  - Local static C variables: stored in either `.bss` or `.data`

```
static int x = 15;

int f() {
    static int x = 17;
    return x++;
}

int g() {
    static int x = 19;
    return x += 14;
}

int h() {
    return x += 27;
}
```

static-local.c

Compiler allocates space in `.data` for each definition of `x`

Creates local symbols in the symbol table with unique names, e.g., `x`, `x.1721` and `x.1724`.

What if you mess up?

```
int x=7;
p1() {}
```

```
extern int x;
p2() {}
```

Correct program.
Only one definition of **`x, p1, p2`**

```
int x=7;
p1() {}
```

```
int x=0;
p1() {}
```

Link error: two definitions of **`x`** and **`p1`**

```
int x;
p1() {}
```

```
int x;
p2() {}
```

Compiler-dependent. Might be considered either one or two definitions of **`x`**.

```
int x=7;
int y=5;
p1() {}
```

```
extern double x;
p2() {}
```

Undefined behavior. No link error.
Writes to **`x`** in **`p2`** may overwrite **`y`**!

```
char p1[]
  = 0xC3;
```

```
extern void p1();
p2() { p1(); }
```

Undefined behavior. No link error.
Call to p1 may crash!

Linker checks for two definitions of one symbol.
Linker *does not* check types of references.

Type Mismatch Example

```
extern long int x;

int main(int argc,
         char *argv[]) {
    printf("%ld\n", x);
    return 0;
}
```

mismatch-main.c

```
double x = 3.14;
```

mismatch-variable.c

- **Compiles without any errors or warnings**
- **What gets printed?**

```
-bash-4.2$ ./mismatch
4614253070214989087
```

Detecting the Type Mismatch Example

```
extern long int x;
```
mismatch.h

```
#include "mismatch.h"

int main(int argc,
         char *argv[]) {
    printf("%ld\n", x);
    return 0;
}
```
mismatch-main.c

```
#include "mismatch.h"

double x = 3.14;
```
mismatch-variable.c

- **Now we get an error … from the *compiler*, not the linker.**

```
mismatch-variable.c:3:8: conflicting types for 'x'
mismatch.h:1:17: previous declaration of 'x'
```

Rules for avoiding type mismatches

- **Avoid global variables as much as possible**
- **Use `static` as much as possible**
- **Declare *everything* that's not `static` in a header file**
  - Make sure to include the header file everywhere it's relevant
  - Including the files that define those symbols
- **Always put `extern` on declarations in header files**
  - Unnecessary but harmless for function declarations
  - Avoids the quirky behavior of extern-less global variables
- **Always write (void) when a function takes no args**
  - `extern void no_args(void);`
  - Leaving out the **`void`** means "I'm *not saying* what argument list this function takes." Turns off argument type checking!

What Do Linkers Do? (cont'd)

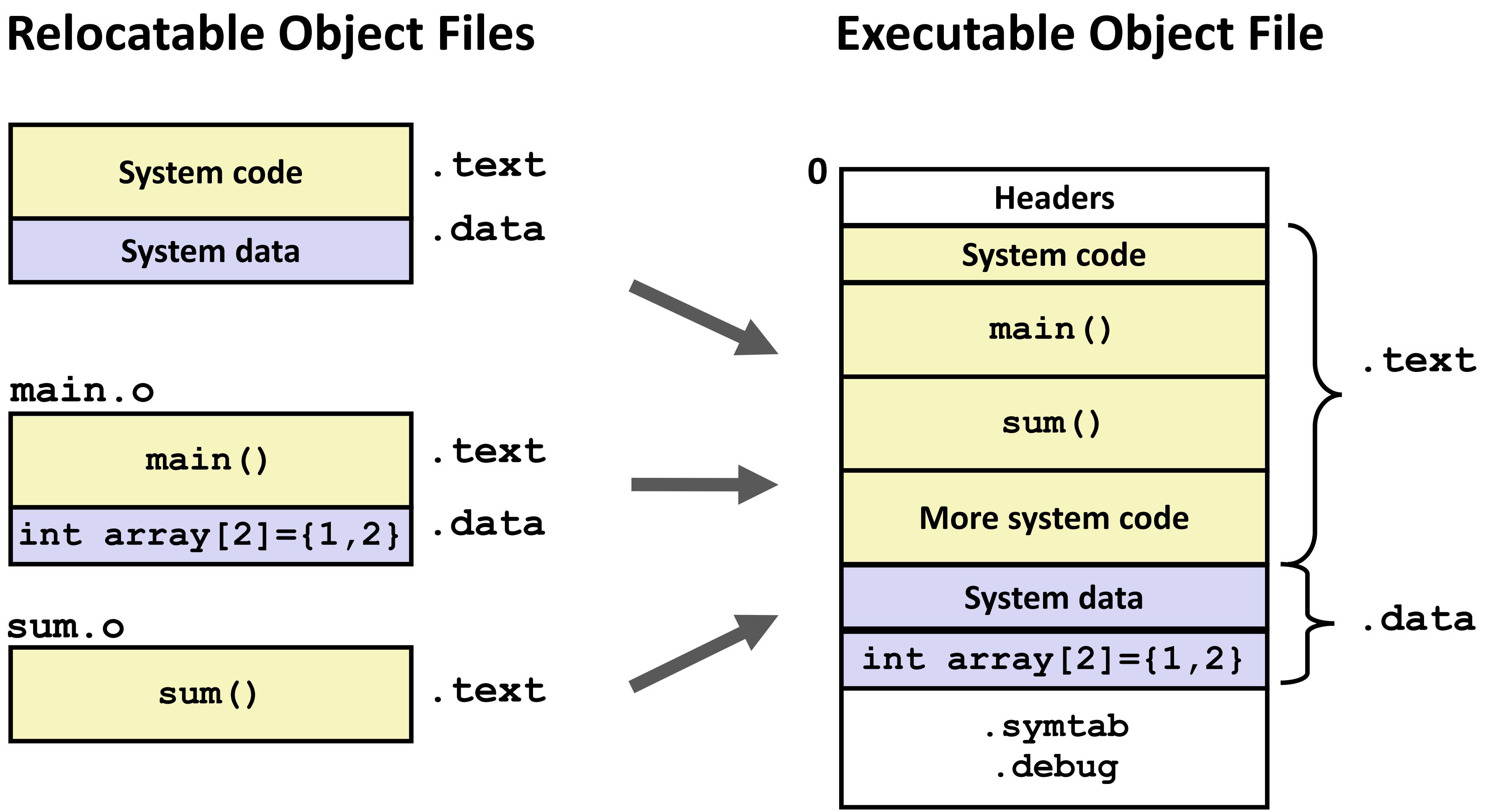
- **Step 2: Relocation**
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.
  - Updates all references to these symbols to reflect their new positions.

Linking Example

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc,char **argv)
{
    int val = sum(array, 2);
    return val;
}
```
main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```
sum.c

Step 2: Relocation



Relocated .text section

```
00000000004004d0 <main>:
  4004d0:       48 83 ec 08          sub    $0x8,%rsp
  4004d4:       be 02 00 00 00       mov    $0x2,%esi
  4004d9:       bf 18 10 60 00       mov    $0x601018,%edi  # %edi = &array
  4004de:       e8 05 00 00 00       callq  4004e8 <sum>    # sum()
  4004e3:       48 83 c4 08          add    $0x8,%rsp
  4004e7:       c3                   retq

00000000004004e8 <sum>:
  4004e8:       b8 00 00 00 00             mov    $0x0,%eax
  4004ed:       ba 00 00 00 00             mov    $0x0,%edx
  4004f2:       eb 09                      jmp    4004fd <sum+0x15>
  4004f4:       48 63 ca                   movslq %edx,%rcx
  4004f7:       03 04 8f                   add    (%rdi,%rcx,4),%eax
  4004fa:       83 c2 01                   add    $0x1,%edx
  4004fd:       39 f2                      cmp    %esi,%edx
  4004ff:       7c f3                      jl     4004f4 <sum+0xc>
  400501:       f3 c3                      repz retq
```

`callq` instruction uses PC-relative addressing for sum():

`0x4004e8` = `0x4004e3` + `0x5`

Source: `objdump -d prog`

Libraries: Packaging a Set of Functions

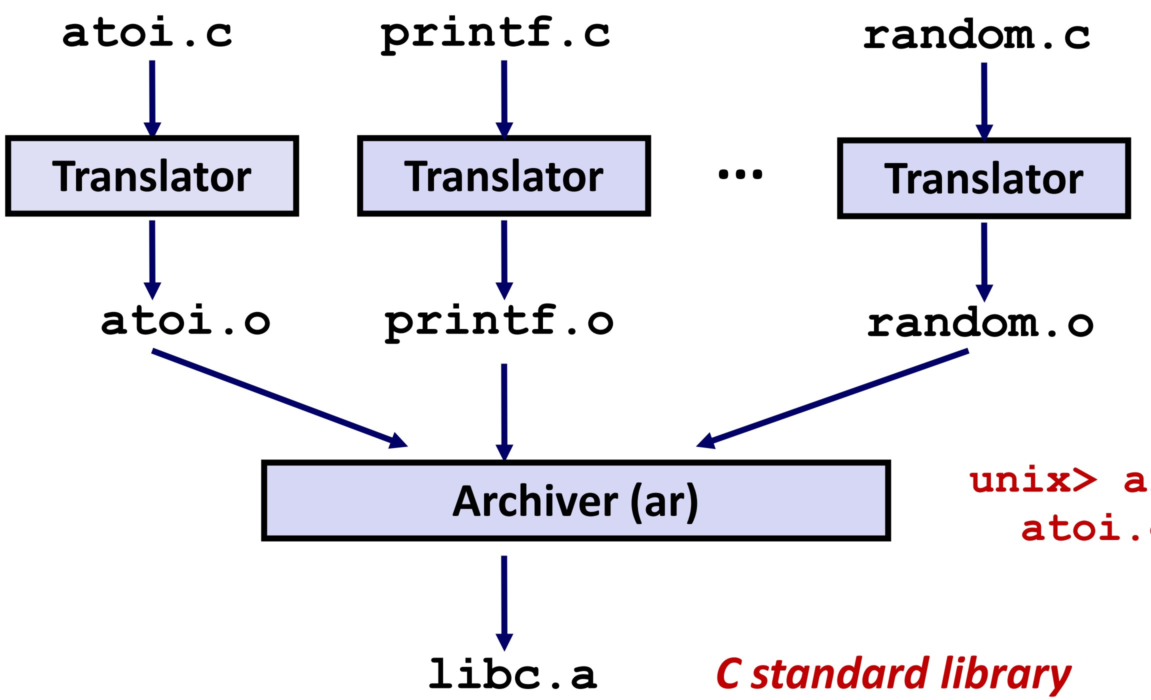
- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.

- **Awkward, given the linker framework so far:**
  - **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

Old-Fashioned Solution: Static Libraries

- **Static libraries (.a archive files)**
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link it into the executable.

Creating Static Libraries




```
unix> ar rs libc.a \
  atoi.o printf.o … random.o
```

- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

Commonly Used Libraries

`libc.a` (the C standard library)

- 4.6 MB archive of 1496 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

`libm.a` (the C math library)

- 2 MB archive of 444 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Linking with Static Libraries

```
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main(int argc, char**
argv)
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n”,
           z[0], z[1]);
    return 0;
}
```

main2.c

libvector.a

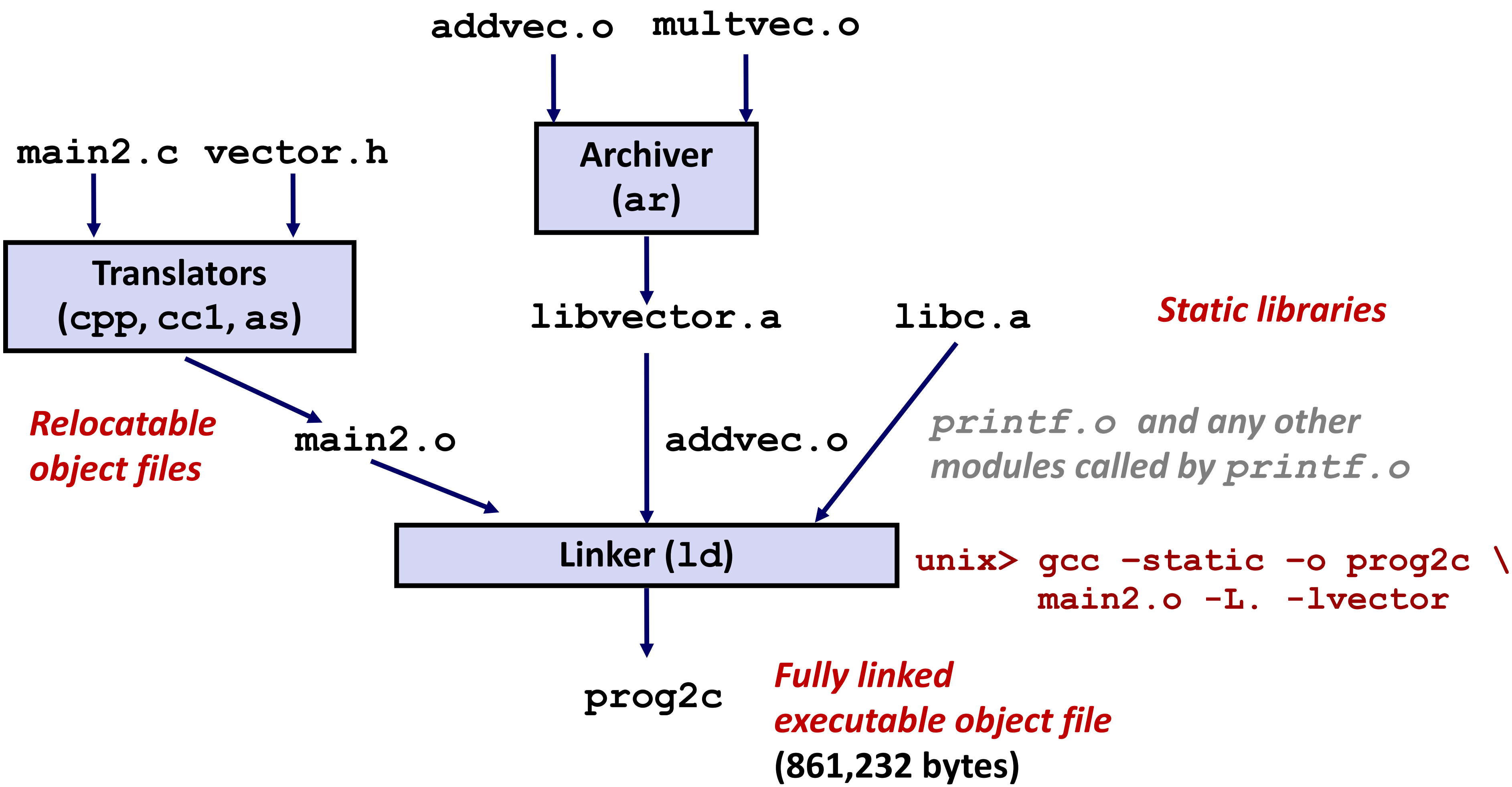
```
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

addvec.c

```
void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

multvec.c

Linking with Static Libraries



"c" for "compile-time"

Using Static Libraries

- **Linker's algorithm for resolving external references:**
  - Scan `.o` files and `.a` files in the command line order.
  - During the scan, keep a list of the current unresolved references.
  - As each new `.o` or `.a` file, *obj*, is encountered, try to resolve each unresolved reference in the list against the symbols defined in *obj*.
  - If any entries in the unresolved list at end of scan, then error.

- **Problem:**
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

```
unix> gcc -static -o prog2c -L. -lvector main2.o
main2.o: In function `main':
main2.c:(.text+0x19): undefined reference to `addvec'
collect2: error: ld returned 1 exit status
```

Quiz Time!

Check out:

https://canvas.cmu.edu/courses/24383/quizzes/67220

If we have time…

- Branch prediction
- Dynamic libraries

What About Branches?

- **Challenge**
  - Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy

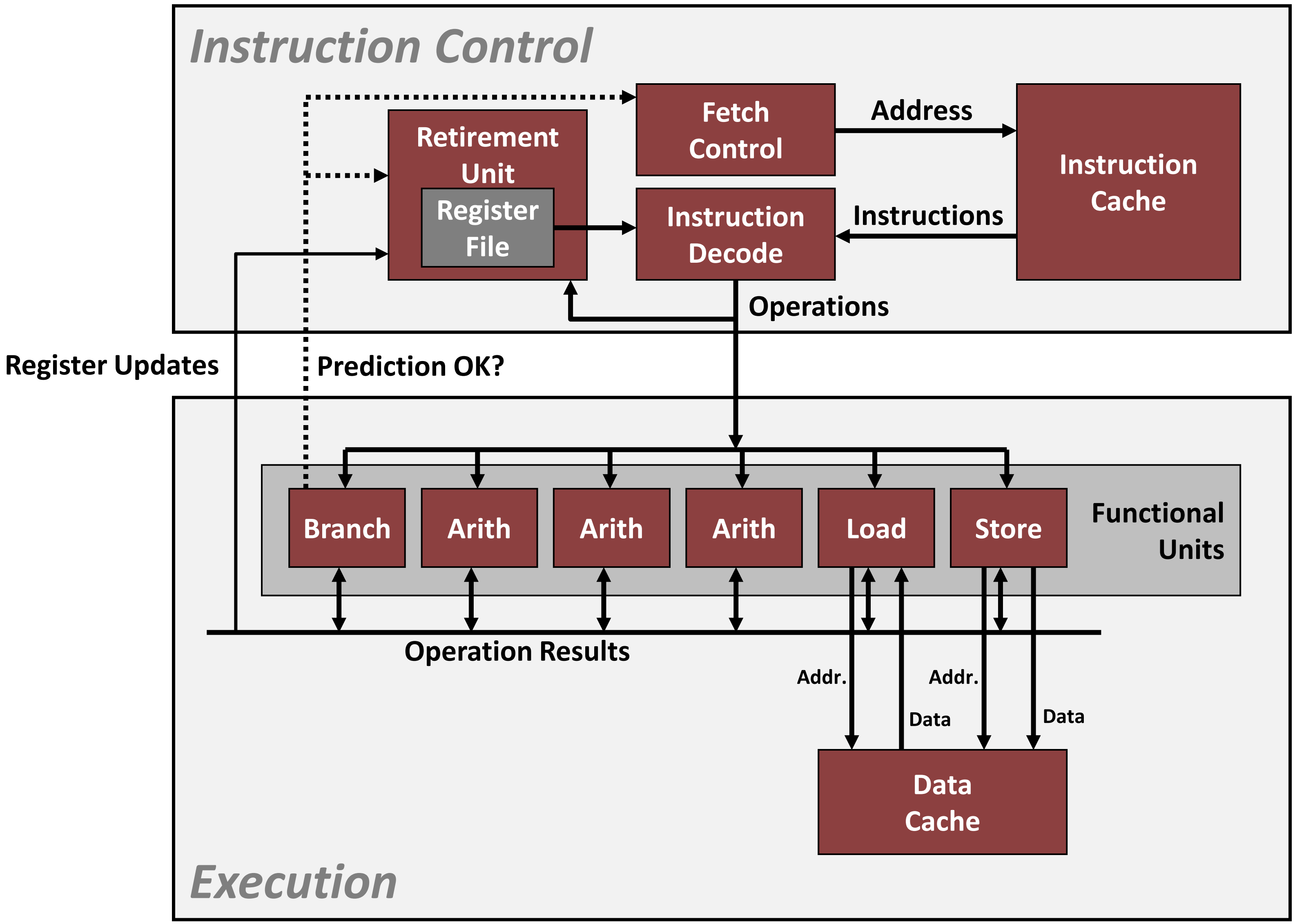
```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax

 . . .

404685:  repz retq
```

Executing

How to continue?

  - When encounters conditional branch, cannot reliably determine where to continue fetching

Modern CPU Design



Branch Outcomes

- **When encounter conditional branch, cannot determine where to continue fetching**
  - Branch Taken: Transfer control to branch target
  - Branch Not-Taken: Continue with next instruction in sequence
- **Cannot resolve until outcome determined by branch/integer unit**

```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax

 . . .

404685:  repz retq
```

Branch Not-Taken

Branch Taken

Branch Prediction

- **Idea**
  - Guess which way branch will go
  - Begin executing instructions at predicted position
    - But don't actually modify register or memory data

```
404663:   mov     $0x0,%eax
404668:   cmp     (%rdi),%rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi),%rax

  . . .

404685:   repz retq
```

Predict Taken

Begin Execution

Branch Prediction Through Loop

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```
i = 98

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```
i = 99

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```
i = 100

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```
i = 101

Assume
vector length = 100

Predict Taken (OK)

Predict Taken (Oops)

Read invalid location

Executed

Fetched

Branch Misprediction Invalidation

Assume

vector length = 100

```
401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029
```

i = 98

Predict Taken (OK)

```
401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029
```

i = 99

Predict Taken (Oops)

```
401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029
```

i = 100

```
401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029
```

i = 101

Invalidate

Branch Misprediction Recovery

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
401036:   jmp     401040
  . . .
401040:   vmovsd  %xmm0,(%r12)
```

i = 99

Definitely not taken

Reload Pipeline

- **Performance Cost**
  - Multiple clock cycles on modern processor
  - Can be a major performance limiter

Branch Prediction Numbers

- **Default behavior:**
  - Backwards branches are often loops so predict taken
  - Forwards branches are often if so predict not taken

- **Predictors average better than 95% accuracy**
  - Most branches are already predictable.
- **Annual branch predictor contests at top Computer Architecture conferences**
  - https://www.jilp.org/jwac-2/program/JWAC-2-program.htm
  - Winner: 34.1 mispredictions per kilo-instruction (!)

Getting High Performance

- **Good compiler and flags**
- **Don't do anything sub-optimal**
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers: procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)

- **Tune code for machine**
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly

Modern Solution: Shared Libraries

- **Static libraries have the following disadvantages:**
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink
    - Rebuild everything with glibc?
    - https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html

- **Modern solution: shared libraries**
  - Object files that contain code and data that are loaded and linked into an application *dynamically,* at either *load-time* or *run-time*
  - Also called: dynamic link libraries, DLLs, `.so` files

Shared Libraries (cont.)

- **Dynamic linking can occur when executable is first loaded and run (load-time linking)**
  - Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`)
  - Standard C library (`libc.so`) usually dynamically linked
- **Dynamic linking can also occur after program has begun (run-time linking)**
  - In Linux, this is done by calls to the `dlopen()` interface
    - Distributing software
    - High-performance web servers
    - Runtime library interpositioning
- **Shared library routines can be shared by multiple processes**
  - More on this when we learn about virtual memory

What dynamic libraries are required?

- **`.interp` section**
  - Specifies the dynamic linker to use (i.e., **`ld-linux.so`**)
- **`.dynamic` section**
  - Specifies the names, etc of the dynamic libraries to use
  - Follow an example of **`prog`**

```
(NEEDED)                    Shared library: [libm.so.6]
```

- **Where are the libraries found?**
  - Use "**`ldd`**" to find out:

```
unix> ldd prog
  linux-vdso.so.1 =>  (0x00007ffcf2998000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f99ad927000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f99adcef000)
```