Program optimization & Linking I

朱家启 徐梓越 许珈铭

2023.11.13

Optimization (CS:APP Ch. 5)

徐梓越

Generally Useful Optimizations （编译器安全优化）

- Code motion/precomputation
- Strength reduction
- Sharing of common subexpressions

Final Pseudo Code

```
    i := n-1
L5: if i<1 goto L1
    t2 := 0
    t6 := 4
    t19 := i << 2
L4: if t6>t19 goto L2
    t3 := A[t2]
    t7 := A[t6]
    if t3<=t7 goto L3
    A[t2] := t7
    A[t6] := t3
L3: t2 := t2+4
    t6 := t6+4
    goto L4
L2: i := i-1
    goto L5
L1:
```

Instruction Count
Before Optimizations
29 in outer loop
25 in inner loop

Instruction Count
After Optimizations
15 in outer loop
9 in inner loop

- These were **Machine-Independent Optimizations**.
- Will be followed by **Machine-Dependent Optimizations**, including allocating temporaries to registers, converting to assembly code

两个主要的妨碍优化的因素和优化

- 内存别名使用
- 函数调用

Procedure calls 过程调用

```
void lower(char *s)
{
   size_t 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:
}
```

Procedure 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
- Procedure lower could interact with strlen

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

两个主要的妨碍优化的因素和优化

- Memory aliasing

·Two different memory references specify single location

·Easy to have happen in C

·Get in habit of introducing local variables

Removing Aliasing

```
/* Sum rows is 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
```

Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long 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;
}
```

- Move vec_length out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

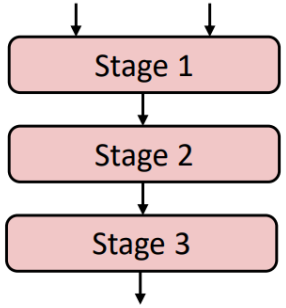
ILP Instruction-level Parallelism 指令级别的并行

- Superscalar Processor
- 数据依赖问题（硬件的特点）

->决定了latency bound

Pipelined Functional Units

```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```



	Time						
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

Haswell CPU

- 8 Total Functional Units
- **Multiple instructions can execute in parallel**
  - 2 load, with address computation
  - 1 store, with address computation
  - 4 integer
  - 2 FP multiply
  - 1 FP add
  - 1 FP divide
- **Some instructions take > 1 cycle, but can be pipelined**

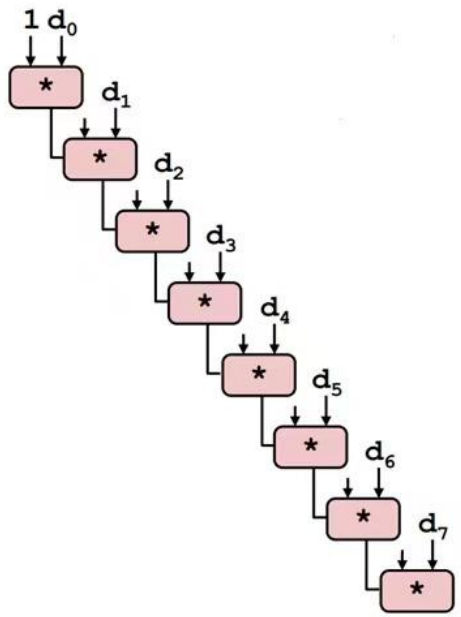
Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	**3-30**	**3-30**
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	**3-15**	**3-15**

打乱bound

Loop Unrolling (2x1)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- **Perform 2x more useful work per iteration**



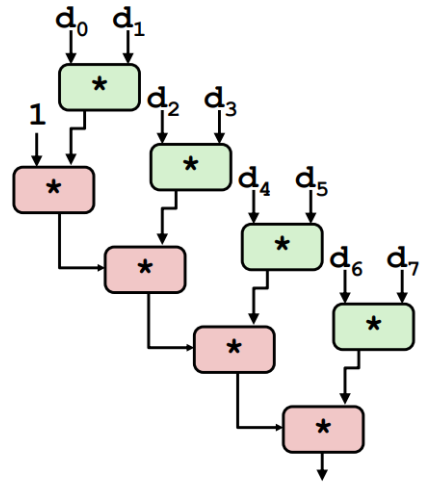
Loop Unrolling with Reassociation (2x1a)

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Compare to before

```
x = (x OP d[i]) OP d[i+1];
```

```
x = x OP (d[i] OP d[i+1]);
```



本质上，我们展开了2个操作，但是还没有完全把顺序关系打断

从Latency Bound 到Throughput Bound

Loop Unrolling with Separate Accumulators (2x2)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
       x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

Haswell CPU

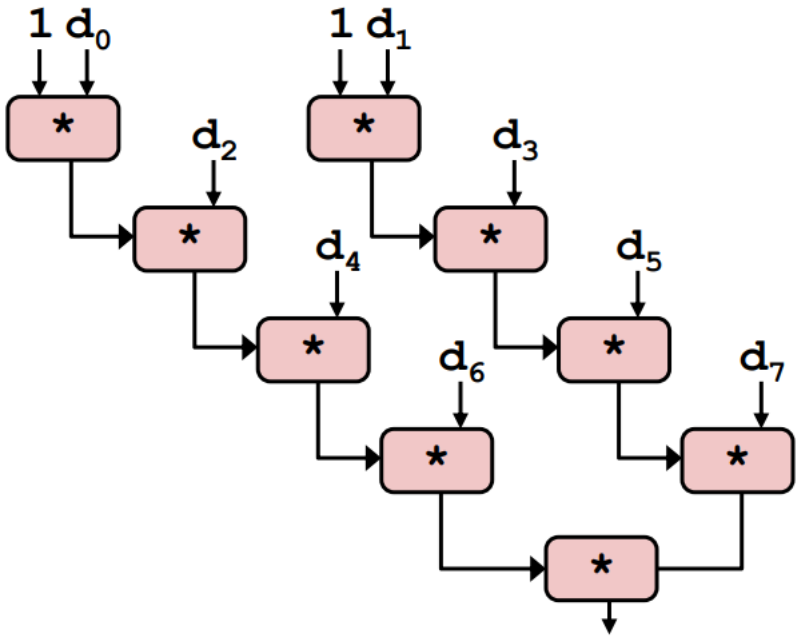
- 8 Total Functional Units

- **Multiple instructions can execute in parallel**
  - 2 load, with address computation
  - 1 store, with address computation
  - 4 integer
  - 2 FP multiply
  - 1 FP add
  - 1 FP divide
- **Some instructions take > 1 cycle, but can be pipelined**

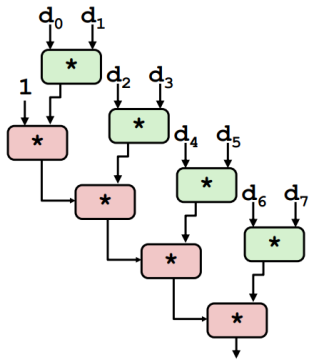
Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	**3-30**	**3-30**
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	**3-15**	**3-15**

Throughput bound

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```




```
x = x OP (d[i] OP d[i+1]);
```



Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	*1.00*	*3.00*	*3.00*	*5.00*
Throughput Bound	*0.50*	*1.00*	*1.00*	*0.50*

Unrolling and Accumulation

- a.有多少op 循环展开的问题 L　　(2x1->2x1a)
- b.有多少接口来存，数据依赖关系是不是 K　　(->2x2)

Limitations

- Add units, load units limited 缓存压力
- Large overhead for short length 对于n大小的预测

Unrolling & Accumulating: Double *

- **Case**
  - Intel Haswell
  - Double FP Multiplication
  - Latency bound: 5.00. Throughput bound: 0.50

FP *	Unrolling Factor L							
Accumulators K	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51		2.51		
3			1.67					
4				1.25		1.26		
6					0.84			0.88
8						0.63		
10							0.51	
12								0.52

Unrolling & Accumulating: Int +

- **Case**
  - Intel Haswell
  - Integer addition
  - Latency bound: 1.00. Throughput bound: 1.00

Int +	Unrolling Factor L							
Accumulators K	1	2	3	4	6	8	10	12
1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	
2		0.81		0.69		0.54		
3			0.74					
4				0.69		1.24		
6					0.56			0.56
8						0.54		
10							0.54	
12								0.56

Achievable Performance

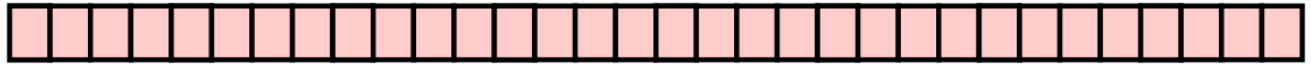
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.52
Latency Bound	*1.00*	*3.00*	*3.00*	*5.00*
Throughput Bound	*0.50*	*1.00*	*1.00*	*0.50*

Achievable performance
CPU的功能部件限制了

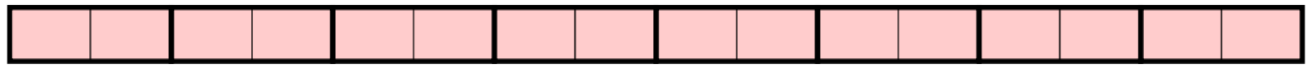
Programming with AVX2

YMM Registers

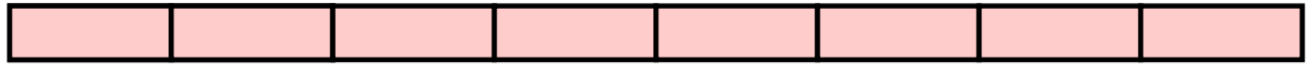
- 16 total, each 32 bytes
- 32 single-byte integers



- 16 16-bit integers



- 8 32-bit integers



- 8 single-precision floats

- 4 double-precision floats

- 1 single-precision float

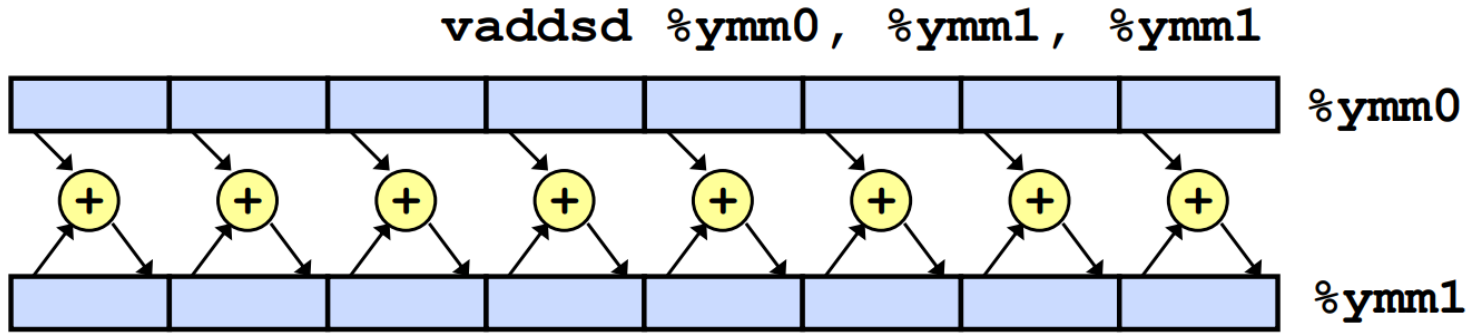
- 1 double-precision float

SIMD Operations
同时做8条单精加法/整型 4条双精
（通常是编译优化 手写的库）

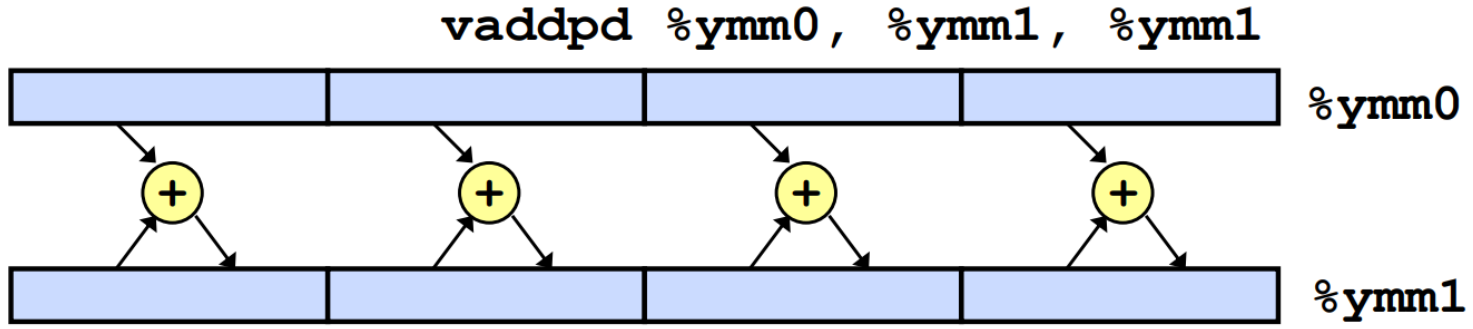
Single Instruction Multiple Data

SIMD Operations

- SIMD Operations: Single Precision



- SIMD Operations: Double Precision



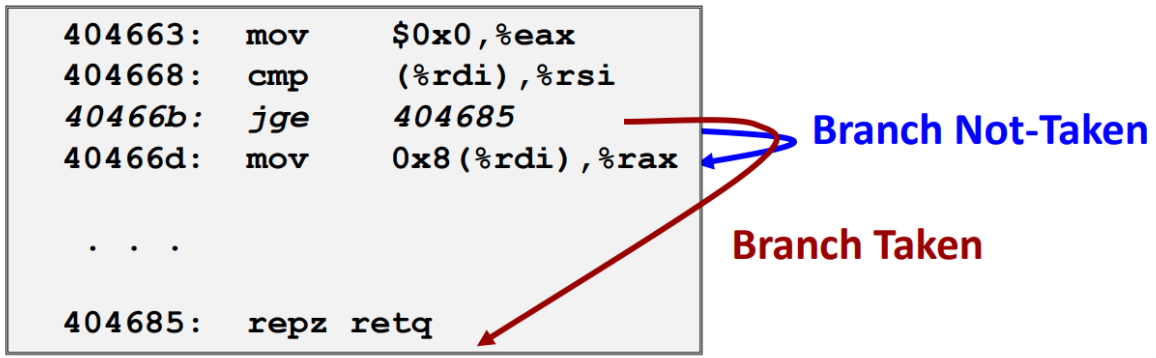
Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
Latency Bound	*0.50*	*3.00*	*3.00*	*5.00*
Throughput Bound	*0.50*	*1.00*	*1.00*	*0.50*
Vec Throughput Bound	*0.06*	*0.12*	*0.25*	*0.12*

Branch

- 限制优化的是Branch 是taken 还是 non-taken
- 通常预测是继续循环，对于一个cycle取2个指令的superscalar

- **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**



Branch Misprediction Invalidation

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

i = 98

Assume
vector length = 100

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

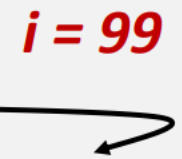
```
401029:   vmulsd (%rdx),%xmm0,%xmm0
40102d:   add    $0x8,%rdx
401031:   cmp    %rax,%rdx
401034:   jne    401029
```

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)
```



Definitely not taken

Reload Pipeline

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

Linking I (CS:APP Ch. 7.1-7.7)

朱家启

- Linker工作概览
- ELF结构
- 符号解析
- 重定位
- 可执行目标文件及其加载

Linker工作概览

(a) `main.c` — *code/link/main.c*

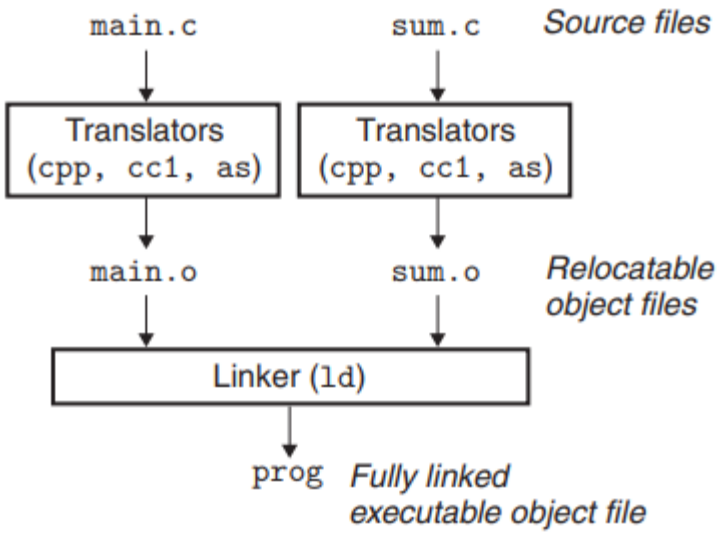
```
1   int sum(int *a, int n);
2
3   int array[2] = {1, 2};
4
5   int main()
6   {
7       int val = sum(array, 2);
8       return val;
9   }
```

code/link/main.c

(b) `sum.c` — *code/link/sum.c*

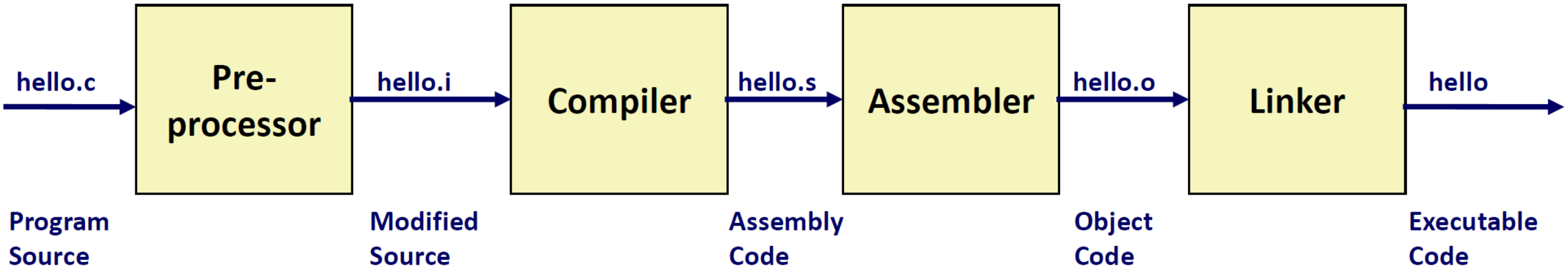
```
1   int sum(int *a, int n)
2   {
3       int i, s = 0;
4
5       for (i = 0; i < n; i++) {
6           s += a[i];
7       }
8       return s;
9   }
```

code/link/sum.c



GCC运行过程（编译器驱动程序）

- Pre-processor(cpp)：展开”#”，只做文本替换，本质是.c变.c
- Compiler(cc1):变成汇编
- Assembler(as):变成二进机器码，带“说明”（ELF文件）
- Linker(ld):主角，组合并变成可执行目标文件（加载器直接复制，不再变化）



Linker工作概览

- 符号解析:
  - 只管“全局”符号（函数内自产自销的不管）
  - 声明、定义、引用都管
- 重定位
  - 把引用改向最终程序的正确位置

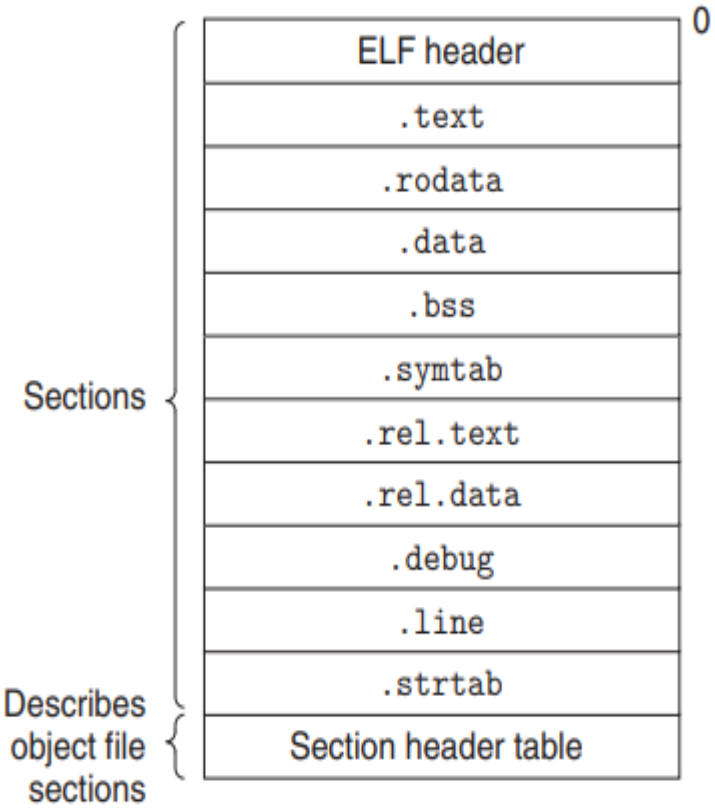
ELF结构

目标文件类型

- 可重定位目标文件(就是前面的.o)：二进制。（linker的食材）
- 共享目标文件（.so)：特殊的前者，允许动态加载。（特种食材）
- 可执行目标文件(a.out)：二进制，可直接复制进内存执行。（做好的菜）

ELF(Executable and Linkable Format)

- ELF header:字长，字节顺序，文件类型，机器类型
- 段头部表：页大小，段内存地址，……
- .text：代码
- .rodata：跳转表，常字符串……
- .data：已初始化的全局变量
- .bss：未初始化的全局变量（不占空间）
- .symtab：符号表
- .rel.text：.text重定位信息（给linker的指示）
- .rel.data：.data重定位信息
- .debug：gcc –g才出现
- 节头部表：每节的偏移和大小
- 注：被处理的.o可以没段头；可执行程序可无节头。



符号表

- Name: Byte offset into string table（存着名字）
- Value: COMMON-对齐限制，.o-节偏移，a.out/.so-虚拟地址
- Size: 数据量，未知为0
- Type: data or function
- Section: 节头部表index
- 理念：字符串数据用索引

code/link/elfstructs.c

```
1  typedef struct {
2      int   name;      /* String table offset */
3      char  type:4,    /* Function or data (4 bits) */
4            binding:4; /* Local or global (4 bits) */
5      char  reserved;  /* Unused */
6      short section;   /* Section header index */
7      long  value;     /* Section offset or absolute address */
8      long  size;      /* Object size in bytes */
9  } Elf64_Symbol;
```

code/link/elfstructs.c

Figure 7.4 **ELF symbol table entry.** The `type` and `binding` fields are 4 bits each.

Linker Symbols（存在.symtab中）

- 全局符号：
  - 自己定义，外部可引用
- 外部符号：也是全局符号
  - 外部定义，自己引用
- 局部符号：
  - 自己定义，自己引用

- Static必然为局部符号
- Static在.bss/.data中
- 链接的“局部”不是程序的“局部”
- 非staic程序变量（自产自销）不存在.symtab中，因为存在栈里

节与伪节(pseudosection)

- 每个符号都分配到某个节
- 三种伪节在节头部表中无条目
- 伪节只在.o中有，a.out没有
- ABS：不该被重定位（main.c)
- UNDEF：未定义（本地引了但本地没定义）
- COMMON：未分配且未初始化

COMMON与.bss区别：背图

	Global Variables	Static Variables
Uninitialized	COMMON	.bss
Initialized to Zero	.bss	.bss
Initialized to Non-Zero	.data	.data

符号解析

符号解析在干啥?

```
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

- 原则：拼好的程序中，每个符号有且只有一个定义
- 每个模块中每个局部符号只有一个定义（编译器检查过了）
- 多个静态局部变量，不能重名，x, x.1, x.2（编译器干过了）
- 跨模块的引用的勾连（链接器干）
- 处理跨模块重名（链接器干）

- 编译器：处理食材 链接器：炒菜

重整

- 同名，但不同参数列表
- Eg. Print()可以是void(int), void(char), void(String)
- 处理办法：多叫几个名字

- 名字命名规则（大致）：bar__3Fooil
- 3：”Foo”这个名字长度为3
- i：第一个参数是int；I：第二个参数是float
- bar: 方法名

解析多重定义的原则

- 符号要么强，要么弱
- 强：函数和已初始化的全局变量
- 弱：未初始化的全局变量（extern可以强制转成弱符号）
- Linux链接器规则：
- 1.（同名字的）强符号最多一个
- 2. 有强有弱，选强的
- 3. 全弱，随机挑

“智力有限”的链接器

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

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

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

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

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

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

- 链接器不管引用变量的类型，这可能会出错
- 解决方案：
- 1.能用static就用static（Recall：static是局部的，冲突只在全局）
- 2.非静态的全放header file，里面全加extern

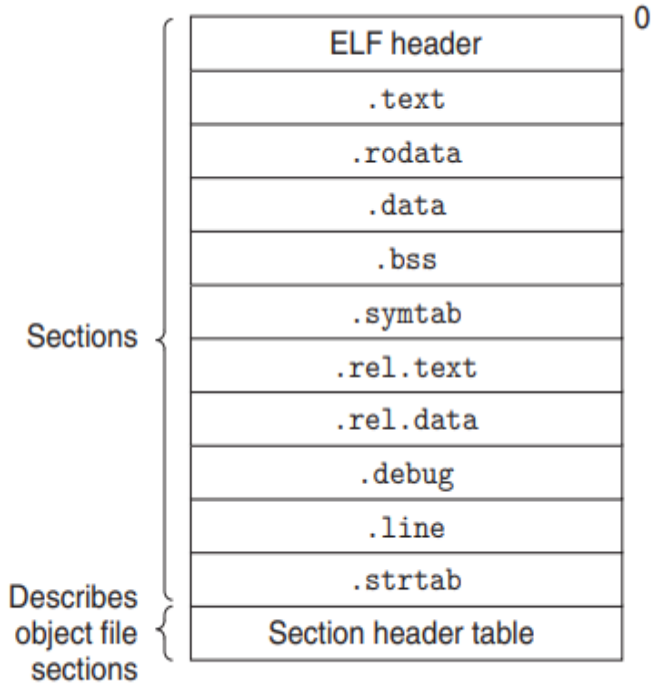
重定位

重定位在干啥?

- “合并同类项”
  - 把所有相同section拼在一起
- “修改路牌”
  - 把所有引用改到最终总程序的正确位置

重定位条目

- 留给链接器的说明，告诉它每个引用该怎么改
- 重定位类型：
- R_X86_64_PC32：用32位PC相对地址访问
- R_X86_64_32：用32位绝对地址访问



code/link/elfstructs.c

```
1   typedef struct {
2       long offset;    /* Offset of the reference to relocate */
3       long type:32,   /* Relocation type */
4            symbol:32; /* Symbol table index */
5       long addend;    /* Constant part of relocation expression */
6   } Elf64_Rela;
```

code/link/elfstructs.c

重定位实例(P480-481)

```
1   foreach section s {
2       foreach relocation entry r {
3           refptr = s + r.offset;  /* ptr to reference to be relocated */
4
5           /* Relocate a PC-relative reference */
6           if (r.type == R_X86_64_PC32) {
7               refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8               *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
9           }
10
11          /* Relocate an absolute reference */
12          if (r.type == R_X86_64_32)
13              *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
14      }
15  }
```

Figure 7.10 **Relocation algorithm.**

code/link/main-relo.d

```
1   0000000000000000 <main>:
2      0:   48 83 ec 08             sub    $0x8,%rsp
3      4:   be 02 00 00 00          mov    $0x2,%esi
4      9:   bf 00 00 00 00          mov    $0x0,%edi        %edi = &array
5                           a: R_X86_64_32 array           Relocation entry

6      e:   e8 00 00 00 00          callq  13 <main+0x13>   sum()
7                           f: R_X86_64_PC32 sum-0x4       Relocation entry
8     13:   48 83 c4 08             add    $0x8,%rsp
9     17:   c3                      retq
```

code/link/main-relo.d

Figure 7.11 **Code and relocation entries from** `main.o`. The original C code is in Figure 7.1.

Practice

朱家启

The End