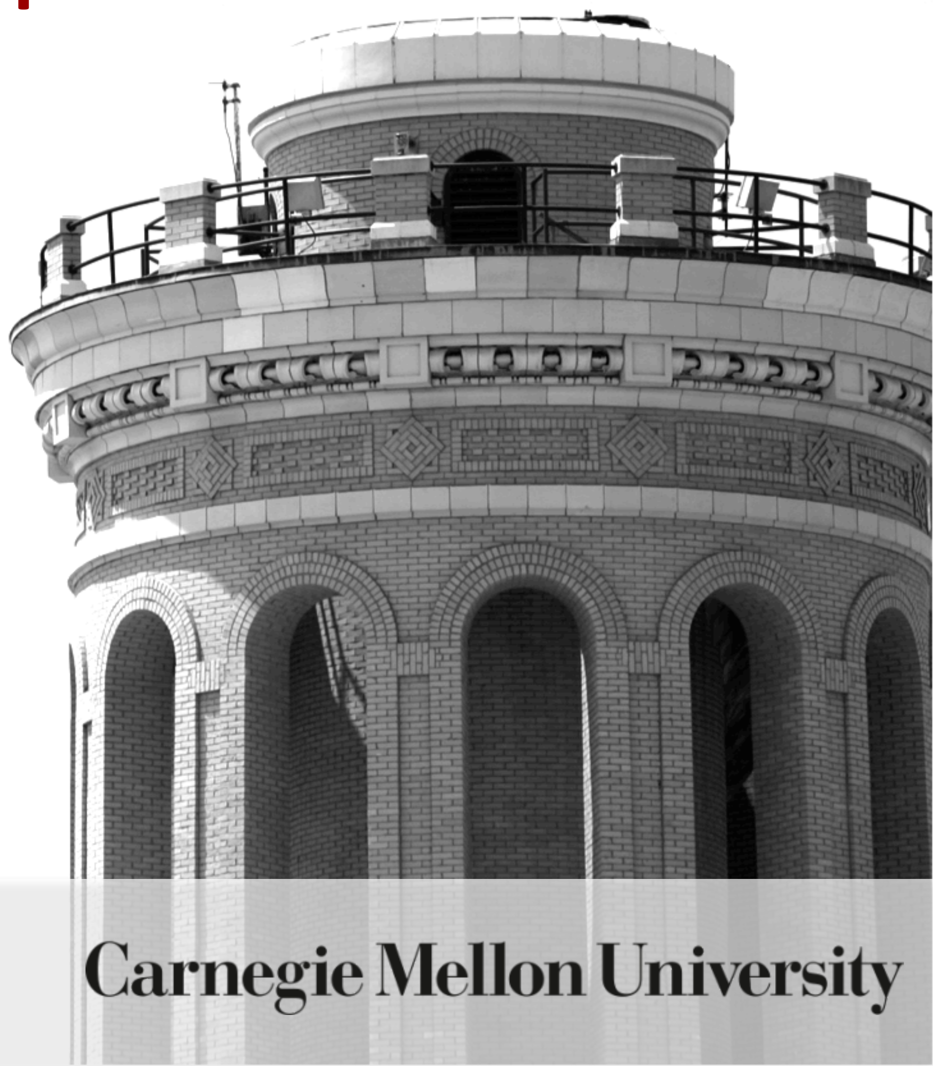
18-349: Introduction to Embedded Real-Time Systems

Lecture 7: Profiling and Optimization

Anthony Rowe
Electrical and Computer Engineering
Carnegie Mellon University



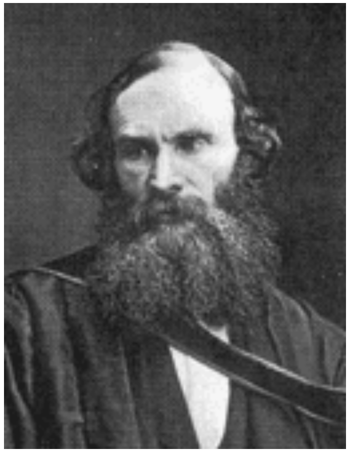
Electrical & Computer ENGINEERING

Carnegie Mellon University

Lecture Overview

- System Profiling
  - Speed
  - Size
  - Power
- ARM Optimization

System Profiling



"If you can not measure it, you can not improve it."

– Lord Kelvin

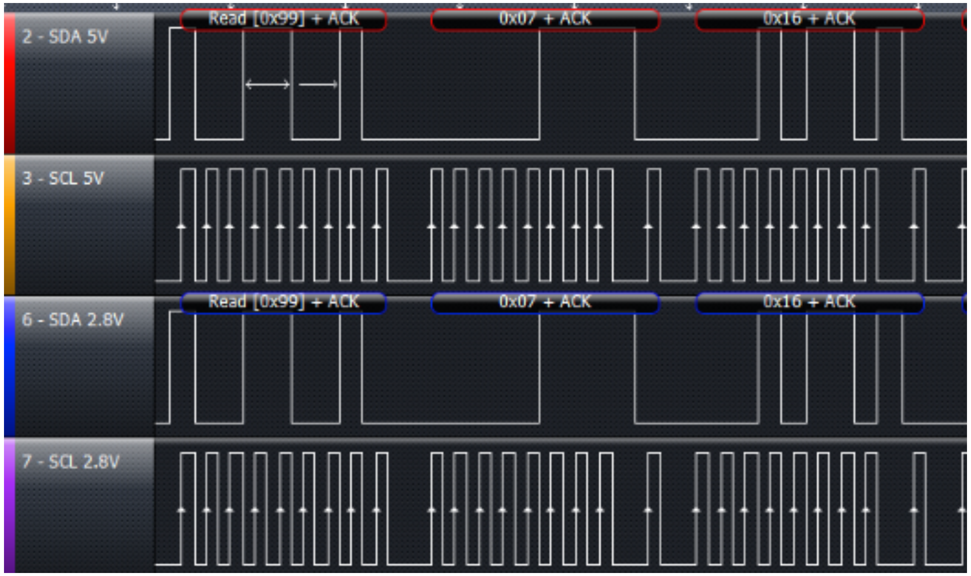
What can we improve?

System Profiling

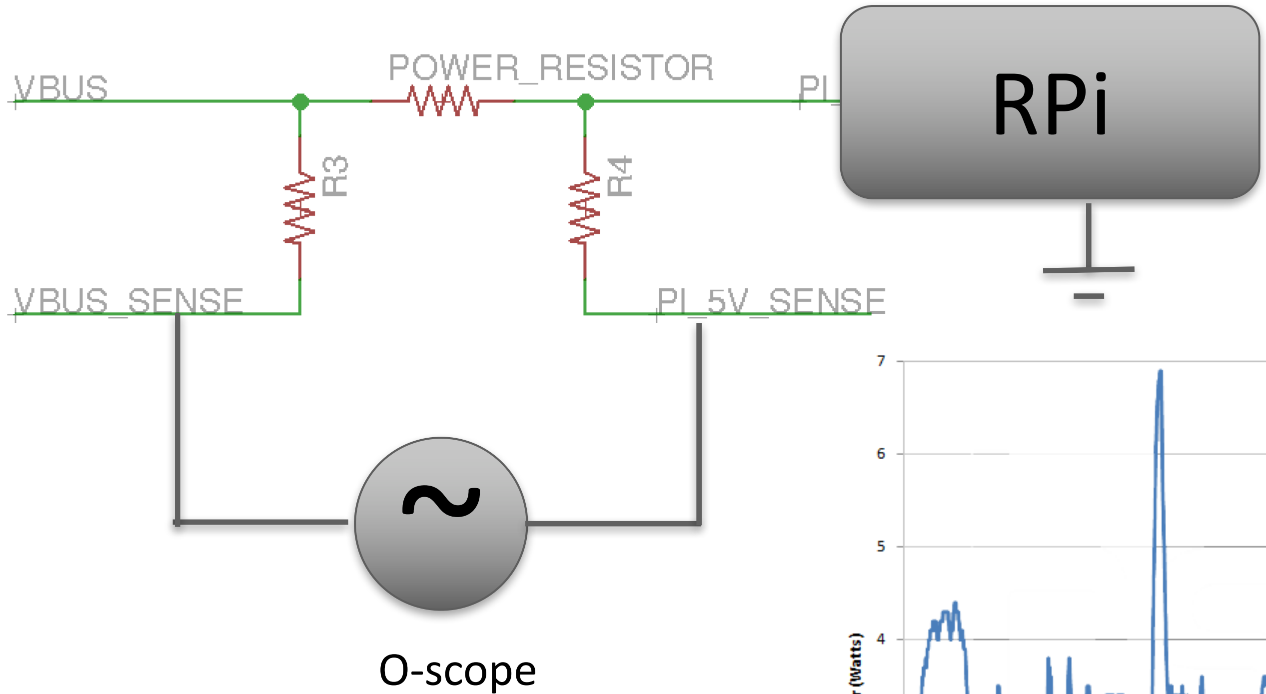
- Algorithm Efficiency
  - Big “O” notation for limit of functions
- Code Speed
  - Measure execution time
- Code Size
  - Number of bytes
- Memory Consumed
- Power Consumed

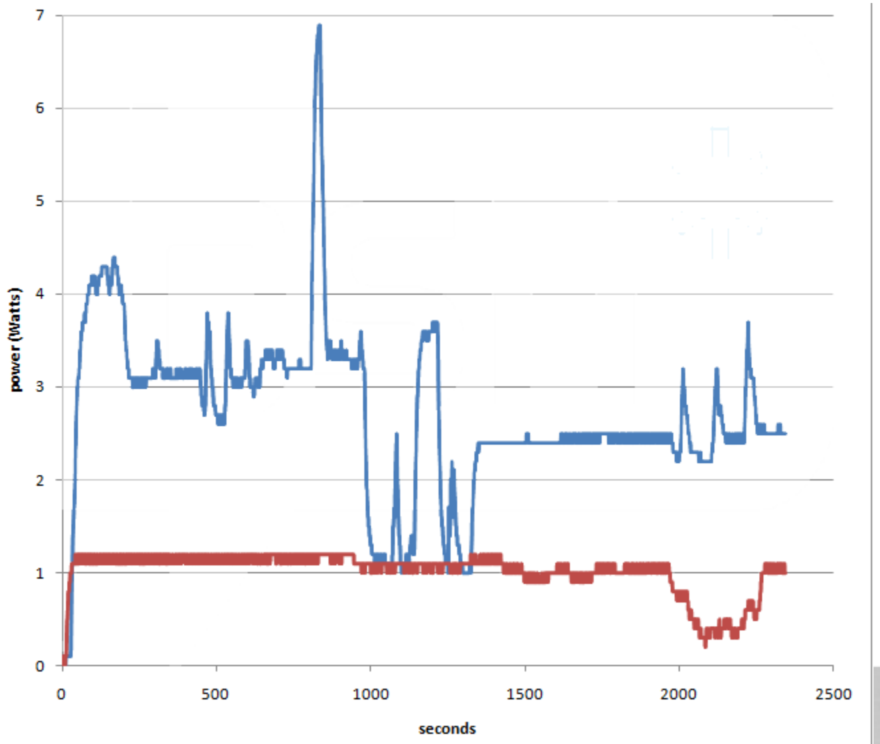
GPIO Timing

- GPIO are often the lowest latency I/O



Energy Profile





Deeper Profiling

- Use timers to benchmark internal code segments
- Get a Linux Runtime Version of *SOME* of your code
  - Port relevant algorithmic components to an existing OS on a less-constrained environment
- Now you can use existing tools
  - Gprof
  - Valgrind

Gprof

- Provides timing information for function call tree
- Built into GNU Binutils
- Instruments code at the start of function calls
  - Inserts mcount function
- Sampling probes PC given OS interrupt calls
  - Requires an OS or at least profile interrupts
- More information: http://docs.freebsd.org/44doc/psd/18.gprof/paper.pdf

Valgrind

- Programming tool(s) for memory debugging, memory leak detection and profiling

- Most common tool is Memcheck
  - Use of uninitialized memory
  - Read/writing freed memory
  - Read/write passed array bound or malloc
  - Memory Leaks

- Many other tools
  - Massif (heap profiler)
  - Helgrind (checks race conditions in threads)
  - Cachegrind (cache profiler)
  - Callgrind (call graph analyser)
  - Exp-sgcheck (experimental stack and global memory tool)

Optimization Overview

- Code Optimization Techniques
  - Processor independent compiler optimizations
    - Common sub-expression elimination
    - Dead-code elimination
    - Induction variables
    - In-lining of functions
    - Loop unrolling
- ARM specific optimization
    - Count-down loops
    - Register spilling
    - Efficient use of global variables
- Space Optimization

Code Optimization

- Embedded systems usually contain a few key functions that determine the system's performance

- By optimizing these functions, you can reduce the number of clock cycles required by a program, and reduce program's power consumption

- Several ways to optimize programs
  - Choose efficient algorithms and data structures
  - Write code that can be optimized by a compiler
  - Convert C code to assembly code

- Code optimization can also involve optimizing the amount of memory used
  - Memory optimization techniques often (but not always) conflict with clock-cycle optimization techniques

Improving Program Performance

- Compiler writers try to apply several standard optimizations
  - Do **not** always succeed
  - Have to ensure that the program will produce the same output for *all* cases
- Optimizations based on specific architecture/implementation characteristics can be very helpful
- How can one help?
  - Reorganize code to help compiler find opportunities for improvement

Processor-Independent Optimizations (1)

- Common Sub-expression Elimination
  - Formally, "An occurrence of an expression E is called a *common sub-expression* if E was previously computed, and the values of variables in E have not changed since the previous computation."
  - You can avoid re-computing the expression if we can use the previously computed one.
  - Benefit: less code to be executed

BEFORE

```
b:
    t6  = 4 * i
    x   = a[t6]
    t7  = 4 * i
    t8  = 4 * j
    t9  = a[t8]
    a[t7] = t9
    t10 = 4 * j
    a[t10] = x
    goto b
```

AFTER

```
b:
    t6  = 4 * i
    x   = a[t6]
    t8  = 4 * j
    t9  = a[t8]
    a[t6] = t9
    a[t8] = x
    goto b
```

Processor Independent Optimizations (2)

- Dead-Code Elimination
  - If code is definitely *not* going to be executed during *any* run of a program, then it is called dead-code and can be removed
  - Example:

```
debug = 0;
...
if (debug){
   print .....
}
```

  - **You** can help by using `#ifdefs` to tell the compiler about dead-code
    - It is often difficult for the compiler to identify dead-code itself

Processor Independent Optimizations (3)

- Induction Variables and Strength Reduction
  - A variable X is called an *induction variable* of a loop L if every time the variable X changes value, it is incremented or decremented by some constant
  - When there are 2 or more induction variables in a loop, it may be possible to get rid of all but one
  - It is also frequently possible to perform strength reduction on induction variables
    - The strength of an instruction corresponds to its execution cost
  - Benefit: Fewer and less expensive operations

```
    j = 0
label_XXX
    j  = j + 1
    t4 = 11 * j
    t5 = a[t4]
    if (t5 > v) goto label_XXX
```

```
    t4 = 0
label_XXX
    t4 += 11
    t5 = a[t4]
    if (t5 > v) goto label_XXX
```

Processor-Independent Optimizations (4)

- Loop Unrolling
  - Doing multiple iterations of work in each iteration is called "loop unrolling"
  - Benefit: reduction in looping overheads and opportunity for more code opts.
  - **Danger**: increased code size and *nonintegral loop div*.
  - Appropriate when loops are small

```
int checksum(int *data, int N)
{
        int i, sum=0;
        for(i=0;i<N;i++)
        {
                sum += *data++;
        }
        return sum;
}
```

BEFORE

```
int checksum(int *data, int N)
{
      int i, sum=0;
      for(i=0;i<N;i+=4)
      {
          sum += *data++;
          sum += *data++;
          sum += *data++;
          sum += *data++;
      }
      return sum;
}
```

AFTER

Loop Unrolling

Original loop

BEFORE

```
0x00:    MOV      r3,#0   ; sum =0
0x04:    MOV      r2,#0   ; i= 0
0x08:    CMP      r2,r1   ; (i < N) ?
0x0c:    BGE      0x20    ; go to 0x20 if i >= N
0x10:    LDR      r12,[r0],#4 ; r12 <- data++
0x14:    ADD      r3,r12,r3    ; sum = sum + r12
0x18:    ADD      r2,r2,#1     ; i=i+1
0x1c:    B        0x8     ; jmp to 0x08
0x20:    MOV      r0,r3   ; sum = r3
0x24:    MOV      pc,r14  ; return
```

loop overhead
computed N times

After unrolling
the loop 4 times

AFTER

```
0x00:    MOV      r3,#0    ; sum = 0
0x04:    MOV      r2,#0    ; i = 0
0x08:    B        0x30     ; jmp to 0x30
0x0c:    LDR      r12,[r0],#4  ; r12 <- data++
0x10:    ADD      r3,r12,r3    ; sum = sum + r12
0x14:    LDR      r12,[r0],#4  ; r12 <- data++
0x18:    ADD      r3,r12,r3    ; sum = sum + r12
0x1c:    LDR      r12,[r0],#4  ; r12 <- data++
0x20:    ADD      r3,r12,r3    ; sum = sum + r12
0x24:    LDR      r12,[r0],#4  ; r12 <- data++
0x28:    ADD      r3,r12,r3    ; sum = sum + r12
0x2c:    ADD      r2,r2,#4     ; i = i + 4
0x30:    CMP      r2,r1    ; (i < N) ?
0x34:    BLT      0xc      ; go to 0x0c if i < N
0x38:    MOV      r0,r3    ; r0 <- sum
0x3c:    MOV      pc,r14   ; return
```

loop overhead
computed N/4 times

Processor-Independent Compiler Optimizations (5)

- In-lining of functions
  - Replacing a call to a function with the function's code is called "in-lining"
  - Benefit: reduction in procedure call overheads and opportunity for additional code optimizations
  - Danger: increased code size (possibly)
  - Appropriate when small and/or called from a small number of sites

```
void t(int x, int y)
{
    int a1=max(x,y);
    int a2=max(x+1,y);

    return max(a1+1,a2);
}
__inline int max(int a, int b)
{
    int x;
    x=(a>b ? a:b);
    return x;
}
```

Without Function Inlining

```
void t(int x, int y)
{
    int a1=max(x,y);
    int a2=max(x+1,y);

    return max(a1+1,a2);
}
int max(int a, int b)
{
    int x;
    x=(a>b ? a:b);
    return x;
}
```

```
max
$a
0x00:    CMP     r0,r1;    (x > y) ?
0x04:    BGT     0x0c;     return if (x > y)
0x08:    MOV     r0,r1;    else r0 <- y
0x0c:    MOV     pc,r14    return
t
0x10:    STMFD   r13!,{r4,r14}; save registers
0x14:    MOV     r2,r0;         r2 <- x
0x18:    MOV     r3,r1;         r3 <- y
0x1c:    MOV     r1,r3;         r1 <- y
0x20:    MOV     r0,r2;         r0 <- x
0x24:    BL      max  ;         r0 <- max(x,y)
0x28:    MOV     r4,r0;         r4 <- a1
0x2c:    MOV     r1,r3;         r1 <- y
0x30:    ADD     r0,r2,#1;      r0 <- x+1
0x34:    BL      max  ;             r0 <- max(x+1,y)
0x38:    MOV     r1,r0 ;             r1 <- a2
0x3c:    ADD     r0,r4,#1 ;     r0 <- a1+1
0x40:    LDMFD   r13!,{r4,r14} ; restore
0x44:    B       max  ;
```

With Function Inlining

```
void t(int x, int y)
{
    int a1=max(x,y);
    int a2=max(x+1,y);

    return max(a1+1,a2);
}
__inline int max(int a, int b)
{
    int x;
    x=(a>b ? a:b);
    return x;
}
```

```
0x00:    CMP    r0,r1 ; (x<= y) ?
0x04:    BLE    0x10  ; jmp to 0x10 if true
0x08:    MOV    r2,r0 ; a1 <- x
0x0c:    B      0x14  ; jmp to 0x14
0x10:    MOV    r2,r1 ; a1 <- y if x <= y
0x14:    ADD    r0,r0,#1; generate r0=x+1
0x18:    CMP    r0,r1   ;    (x+1 > y) ?
0x1c:    BGT    0x24    ;jmp to 0x24 if true
0x20:    MOV    r0,r1   ; r0 <- y
0x24:    ADD    r1,r2,#1 ; r1 <- a1+1
0x28:    CMP    r1,r0 ; (a1+1 <= a2) ?
0x2c:    BLE    0x34  ; jmp to 0x34 if true
0x30:    MOV    r0,r1 ; else r0 <- a1+1
0x34:    MOV    pc,r14
```

Negative Instruction-Cache Effects

- Negative instruction-cache effects
  - Loop unrolling and function in-lining can cause performance degradation in systems with caches

```
int checksum(int *data, int N)
{
        int i;
        for(i=N;i>=0;i--)
        {
          sum += *data++;
        }
        return sum;
}
```

```
0x0000000c:    LDR    r3,[r2],#4
0x00000010:    ADD    r0,r3,r0
0x00000014:    SUB    r1,r1,#1
0x00000018:    CMP    r1,#0
0x0000001c:    BGE    0xc
```

Before unrolling the loop

0x0000000C
0x00000010
0x00000014
0x00000018
0x0000001C

instruction cache

Negative Instruction-Cache Effects (contd)

After unrolling the loop

```
int checksum(int *data, int N)
{
        int i;
        for(i=N;i>=0;i-=4)
        {

            sum += *data++;
            sum += *data++;
            sum += *data++;
            sum += *data++;

        }


return sum;
}
```

```
0x00000008:  LDR   r3,[r0],#4
0x0000000c:  ADD   r2,r3,r2
0x00000010:  LDR   r3,[r0],#4
0x00000014:  ADD   r2,r3,r2
0x00000018:  LDR   r3,[r0],#4
0x0000001c:  ADD   r2,r3,r2
0x00000020:  LDR   r3,[r0],#4
0x00000024:  ADD   r2,r3,r2
0x00000028:  SUB   r1,r1,#4
0x0000002c:  CMP   r1,#0
0x00000030:  BGE   0x8
```

instruction cache

ARM-Specific Code-Optimization Techniques

- Often, it is important to understand the architecture's implementation in order to effectively optimize code
- One example of this is the ARM barrel shifter
  - Can convert Y * Constant into series of adds and shifts
    - Y * 9 = Y * 8 + Y * 1
    - Assume R1 holds Y and R2 will hold the result
      - ADD R2, R1, R1, LSL #3 ; LSL #3 is same as * by 8
- Use of conditional execution of instructions can reduce the code size as well as reduce the number of execution cycles

Writing Efficient C for ARM Processors (1)

- Use loops that count down to zero, instead of counting upwards
- Example

```
int checksum(int *data)
{
   unsigned i;
   int sum=0;

   for(i=0;i<64;i++)
    sum += *data++;

   return sum;
}
```

Count-up loop

```
int checksum (int *data)
{
   unsigned i;
   int sum=0;

   for(i=63;i >= 0;i--)
    sum += *data++;

   return sum;
}
```

Count-down loop

- Counting upwards needs an `ADD` instruction, a `CMP` to check if index less than 64, and a conditional branch if index is less than 64
  - Counting downwards needs a SUBS instruction (which sets the CPSR flags), and a conditional branch instruction BGE to handle the end of the looping

Count-Down Loops (Example)

```
int checksum_v1(int *data)
{
   unsigned i;
   int sum=0;

   for(i=0;i<64;i++)
    sum += *data++;

   return sum;
}
```

```
int checksum_v2(int *data)
{
   unsigned i;
   int sum=0;

   for(i=63;i >= 0;i--)
    sum += *data++;

   return sum;
}
```

```
    MOV  r2, r0; r2=data
    MOV  r0, #0; sum=0
    MOV  r1, #0; i=0
L1  LDR  r3,[r2],#4;   r3=*data++
    ADD  r1, r1, #1;   i=i+1
    CMP  r1, 0x40;     cmp r1, 64
    ADD  r0, r3, r0;   sum +=r3
    BCC  L1;       if i < 64, goto L1
    MOV  pc, lr;   return sum
```

```
    MOV  r2, r0; r2=data
    MOV  r0, #0; sum=0
    MOV  r1, #0x3f;  i=63
L1  LDR  r3,[r2],#4;   r3=*data++
    ADD  r0, r3, r0;   sum +=r3
    SUBS  r1, r1, #1;  i--, set flags
    BGE  L1;  if i >= 0, goto L1
    MOV  pc, lr;       return sum
```

Writing Efficient C for ARM Processors (2)

- These are things you can keep in mind, rather than expecting the compiler to do all the work for you
- ARM processors uses 32-bit data types in their data processing instructions
  - If you use types like `char`, the compiler has to add extra code to check/ensure that the value does not exceed 255
- Example

```
void t3(void)
{
    char c;
    int x=0;
    for(c=0;c<63;c++)
        x++;
}
```

```
void t4(void)
{
    int c;
    int x=0;
    for(c=0;c<63;c++)
        x++;
}
```

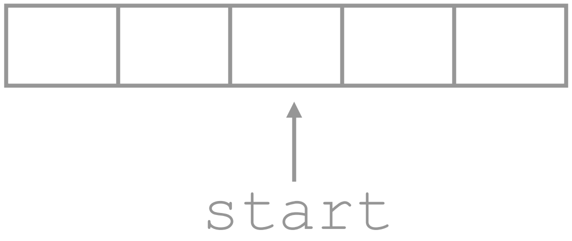
```
     MOV     r0,#0;       x=0
     MOV     r1,#0;       c=0
L1   CMP     r1,#0x3f;    cmp c with 63
     BCS     L2;      if c>= 63, goto L2
     ADD     r0,r0,#1;    x++;
     ADD     r1,r1,#1;    c++
     AND     r1,r1,#0xff; c=(char) r1
     B       L1;      branch to L1
L2   MOV     pc,r14
```

Writing Efficient C for ARM Processors (3)

- ARM does not have a divide instruction
- Divisions are implemented by calling software routines in C library
- Can take between 20-100 cycles

- In many cases, it might be possible to avoid divisions and/or remainder operation
  Example: Circular Buffers (assuming `increment <= size`)

```
start=(start+increment) % size
```




```
 start+= increment;
if (start >= size)
     start -= size;
```

Writing Efficient C for ARM Processors (4)

- Efficiently using global variables: Global variables are stored in memory, load and store instructions are typically used to access the variable when they are used or modified
  - Register accesses are more efficient than memory accesses
- In some cases a global variable is used frequently, it may be better to store it in a local variable
- Example

```
int f(void);
int g(void);
int errs;

void test_v1(void)
{
   errs += f();
   errs += g();
}
```

```
int f(void);
int g(void);
int errs;

void test_v2(void)
{
   int local_errs=errs;
   local_errs += f();
   local_errs += g();
   errs=local_errs;
}
```

Efficient Use of Global Variables

```
test_v1
 0x00:   STMFD   r13!,{r4,r14}       ; save registers
 0x04:   BL      f              ; compute f()
 0x08:   LDR     r4,0x84          ; r4 <- address of errs
 0x0c:   LDR     r1,[r4,#0]          ; r1 <- errs
 0x10:   ADD     r0,r0,r1         ; r1 <- r1 + r0
 0x14:   STR     r0,[r4,#0]       ; store r1 at mem loc address of errs
 0x18:   BL      g              ; compute g()
 0x1c:   LDR     r1,[r4,#0]       ; r1 <- errs
 0x20:   ADD     r0,r0,r1
 0x24:   STR     r0,[r4,#0]       ; store r0 in errs
 0x28:   LDMFD   r13!,{r4,pc}        ; exit from function
```

```
test_v2
 0x00:   STMFD   r13!,{r3-r5,r14}    ; save registers
 0x04:   LDR     r5,0x84             ; r5 <- address of errs
 0x08:   LDR     r4,[r5,#0]          ; r4 = local_errs = errs
 0x0c:   BL      f                ; compute f()
 0x10:   ADD     r4,r0,r4            ; r4 = r4 + f()
 0x14:   BL      g                ; compute g()
 0x18:   ADD     r0,r0,r4            ; r0 = r0 + r4;
 0x1c:   STR     r0,[r5,#0]          ; store r0 at mem loc address of errs
 0x20:   LDMFD   r13!,{r3-r5,pc}     ; exit from function
```

Writing Efficient C for ARM Processors (5)

- Local variables are typically stored in registers
- In some cases, local variables need to be stored in memory
  - Example – when the address of a local variable is taken
  - If a local variable is stored in memory, load and store are used to access the variable
- Example

```
int f(int *a);
int g(int b);

void test_v1(void)
{
   int i=0;
   f(&i);
   i += g(i);
   i += g(i);
   /* lots of access to i */
   return i;
}
```

```
int f(int *a);
int g(int b);

void test_v2(void)
{
   int dummy=0, i;
   f(&dummy);
   i = dummy;
   i += g(i);
   i += g(i);
   /* lots of access to i */
   return i;
}
```

Writing Efficient C for ARM Processors (6)

- Avoid *register spilling*
  - When the number of local variables in use in a function exceeds the number of registers available
  - Causes the compiler to place certain variables in memory
- You should limit the number of live variables in a function
  - Subdividing large functions into multiple small functions may help (keep in mind that there you increase the function call overhead)
  - Use the `register` keyword to tell the compiler which variables have to be stored in registers in case of register spilling

Optimizing Function Calls

- Can the compiler optimize multiple calls to the same function?
- Example: Will the compiler convert

```
void test(int x)
   return(square(x*x) + square(x*x));
```

into

```
void test(int x)
   return(2*square(x*x));
```

?

- Not always, the function `square` can have side-effects

```
int square(int x)
{
   counter++;/* counter is a global variable */
   return(x*x);
}
```

- If the function does not have any side-effects and is defined in the same file as the test function, then the compiler can optimize two calls to `square` into a single call to `square`

Writing Efficient C for ARM Processors (7)

- Pure functions: Function whose output depends only upon the input parameters (and not the value of any other global variables) and do not have any side-effects
- Can tell a compiler that a function is a pure function by using the keyword `__pure` in the declaration of the function
  - This allows the compiler to optimize calls to pure functions regardless of where the function is defined
- Example:

```
__pure int square(int x);

int test(int x)
{
   return (square(x*x) + square(x*x));
}
```

```
__pure int square(int x);

int test(int x)
{
   return (2*square(x*x));
}
```

Optimization for Code Size – Optimizing Structures

- Which of the two structures would be better?

```
struct
{
        char a;
        int b;
        char c;
        short d;
}
```

12 bytes

```
struct
{
        char a;
        char c;
        short d;
        int b;
}
```

8 bytes

More Space Optimization

- Can use the `__packed` key word to instruct the compiler to remove all padding

```
__packed struct
{
        char a;
        int b;
        char c;
        short d;
}
```

8 bytes

- Packed structures are slow and inefficient to access
- ARM Compiler emulates unaligned load and store by using several aligned accesses and using several byte-by-byte operations to get the data
- Use `__packed` only if space is more important than speed and you cannot reduce padding by rearrangement

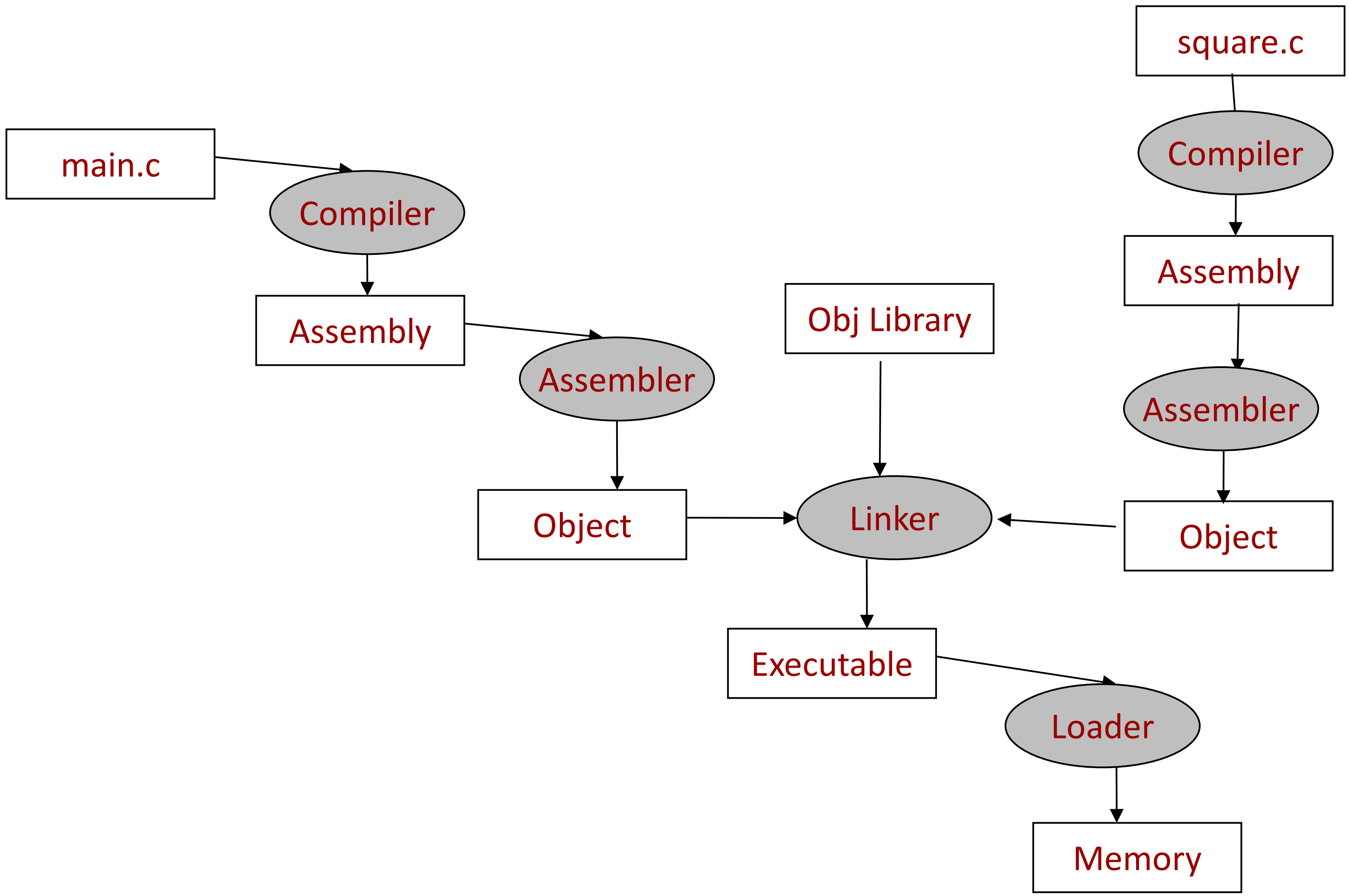
Summary

- Quick SWI / PC Detour
- System Profiling
  - Speed
  - Size
  - Power
- ARM Optimization

Linking Outline

- Linking
  - What happens during linking?
  - How library functions get resolved by the linker?
  - Different kinds of linking
- Executable & Linkable Format
  - ELF header description
  - Description of the sections of an ELF file
- Loading an executable file into memory

Program Translation



Linkers

- Compilers and assemblers generate re-locatable object files
  - References to external symbols are not resolved
  - Compilers generate object files in which code starts at address 0
  - Cannot execute a compiler-produced object file

- Executable files are created from individual object files and libraries through the linking process

- Linker performs two tasks
  - *Symbol resolution*: Object files define and reference symbols, linker tries to resolve each symbol reference with one symbol definition
  - *Relocation*: Linker tries to relocate code and data from different object files so that different sections start at different addresses and all the references are updated

Example: Compiling `main.c` and `square.c`

main.c

```c
int counter=3;
int tmp;
static int sum(int x,
int y);
extern int square(int
x);

int main()
{
        int x=5, y=10;
        int a, b;
        tmp=sum(x,y);
        a=square(x);
        b=square(y);
        tmp=sum(a,b);
        return;
}

int sum(int x, int y)
{
        int result;
        result=x+y;
        return result;
}
```

compiler →

main.o

```
00000000 <sum>:
 0:  add    r0, r0, r1   ; sum=x+y
 4:  bx     lr           ; return
00000008 <main>:
 8:  push   {r4, r5, lr}      ;save registers
 c:  sub    sp, sp, #4        ; sp <- sp- 0x4
10:  mov    r0, #5  ; 0x5     ; x=5;
14:  mov    r1, #10 ; 0xa     ; y=10
18:  bl     0 <sum>           ; compute sum(x,y)
1c : ldr    r5, [pc, #48]     ; r5 <= &tmp (54)
20:  str    r0, [r5]          ; tmp=r0=sum(x,y)
24:  mov    r0, #5    ; x=5;
28:  bl     0 <square>        ; compute  square(5)
2c:  mov    r4, r0            ; r4=r0= 25;
30:  mov    r0, #10 ; 0xa     ; r0=10;
34:  bl     0 <square>        ; compute  square(10)
38:  mov    r1, r0            ; r1=100;
3c:  mov    r0, r4            ; r0=r4=25
40:  bl     0 <sum>           ; compute sum(25, 100)
44:  str    r0, [r5]    ; tmp = r0 = 125;
48:  add    sp, sp, #4        ; sp <= sp + 4
4c:  pop    {r4, r5, lr}      ; restore registers
50:  bx     lr             ; jump back
54:  .word  0x00000000

00000000 <counter>:
 0:  .word  0x00000003    ; address 0x00 of data
section contains 3
```

square.

```c
extern int counter;
int square(int x)
{
  int result;
  if(counter >= 0)
        result=x*x;
  else
        result=0;
  counter--;
  return result;
}
```

compiler ↓

square.

```
0:   ldr    r3, [pc, #32]  ; 28
<square+0x28>
 4:  ldr    r2, [r3]
 8:  cmp    r2, #0  ; 0x0
 c:  movlt  r0, #0  ; 0x0
10:  mulge  r3, r0, r0
14:  movge  r0, r3
18:  sub    r2, r2, #1     ; 0x1
1c:  ldr    r3, [pc, #4]   ; 28
<square+0x28>
20:  str    r2, [r3]
24:  bx     lr
28:  00000000       .word
0x00000000
```

Example: After Linking `main.o` and `square.o`

```
00008338 <sum>:
   8338:    add    r0, r0, r1
   833c:    bx     lr
00008340 <main>:
   8340:    push   {r4, r5, lr}
   8344:    sub    sp, sp, #4     ; 0x4
   8348:    mov    r0, #5  ; 0x5
   834c:    mov    r1, #10 ; 0xa
   8350:    bl     8338 <sum>
   8354:    ldr    r5, [pc, #48]; r5 <= 0x0001056c = &tmp
   8358:    str    r0, [r5]       ; *0x0001056c = tmp = 15
   835c:    mov    r0, #5  ; 0x5
   8360:    bl     8390 <square>  ;
   8364:    mov    r4, r0
   8368:    mov    r0, #10 ; 0xa
   836c:    bl     8390 <square>
   8370:    mov    r1, r0
   8374:    mov    r0, r4
   8378:    bl     8338 <sum>
   837c:    str    r0, [r5]       ; *0x0001056c = tmp = 125
   8380:    add    sp, sp, #4     ;
   8384:    pop    {r4, r5, lr}
   8388:    bx     lr
   838c:    .word  0x0001056c
```

```
00008390 <square>:
   8390: ldr    r3, [pc, #32]  ; r3 = &counter (83b8)
   8394  ldr    r2, [r3]       ; r2 = counter
   8398 cmp     r2, #0  ; 0x0   ; counter > 0 ?
   839c: movlt  r0, #0  ; 0x0   ; if(counter < 0) then
r0<=0x0
   83a0:    mulge  r3, r0, r0   ; else r3 = r0*r0
   83a4:    movge  r0, r3       ; else r0 = r3 =r0 * r0
   83a8:    sub    r2, r2, #1   ; counter--
   83ac:    ldr    r3, [pc, #4] ; r3 = 0x00010564 =
&counter (83b8)
   83b0:    str    r2, [r3]     ; counter = r2 = counter-1
   83b4:    bx     lr           ; return back
   83b8:    .word  0x00010564

00010564 <counter>:
  10564:    .word  0x00000003

.bss
0001056c <tmp>:
  1056c:    .word  0x00000000
```

linker relocates the code to a different memory location

linker adds the actual address of symbol *square*

Library Functions

- What happens when the source files use library functions like `printf`, `scanf`, etc.?
- Compiler produces a symbol (in the same way as the `square` function in the previous example) in the object file
- Linker
  - Attempts to resolve these references by matching them to definitions found in other object files
  - If the symbol is not resolved, the linker searches for the symbol definition in library files
- What are library files?
  - Collection of object files that provide related functionality
  - Example: The standard C library `libc.a` is a collection of object files `printf.o`, `scanf.o`, `fprintf.o, fscanf.o`…

Library Functions

- How does the linker know where to find the library?
  - User defined libraries can be specified as a command line argument
  - The environment variable `LD_LIBRARY_PATH` holds the path that is searched to find the specific library

- Linker does a search to see whether the symbol is defined in the specified libraries
  - The order in which this search is performed is determined by the order in which the libraries are specified
  - If the symbol is defined in more than one library, the first library in the path is selected
    - Linker then extracts the specific .o file that defines the symbol in the library and processes this .o file with all the other object files
  - If the symbol is not defined in any of the library, linker throws an error

Kinds of Linking Models

- Different kinds of linking models
  - ***Static:*** Set of object files, system libraries and library archives are statically bound, references are resolved, and a *self-contained executable file* is created
    - Problem: If multiple programs are running on the processor simultaneously, and they require some common library module (say, `printf.o`), multiple copies of this common module are included in the executable file and loaded into memory (waste of memory!)
  - ***Dynamic:*** Set of object files, libraries, system shared resources and other shared libraries are linked together to create an executable file
    - When this executable is loaded, *other shared resources and dynamic libraries must be made available* in the system for the program to run successfully
    - If multiple programs running on a processor need the same object module, only one copy of the module needs to be loaded in the memory

Dynamic Linking

- Dynamically linked executable or shared object undergoes final linking when
  - Loaded into memory by a program loader
- An executable or shared object to be linked dynamically might
  - List one or more shared objects (shared libraries) with which it should be linked
- Other advantages of dynamic linking
  - Updating of libraries
- The size on disk of an executable that uses dynamically linked modules may be less than its size in memory (during run-time)
  - Why?

Kinds of Object Files

- Three main types of object files
  - **Re-locatable file**: Code and data suitable for linking with other object files to create an executable or a shared object file
  - **Executable file**: Program suitable for execution
  - **Shared object file (also called "Dynamically linked library")**: Special type of re-locatable object file that can be loaded into memory and linked dynamically
    - First, the linker may process it with other re-locatable and shared object files to create another object file
    - Second, the dynamic linker combines it with an executable file and other shared objects to create a process image
- Compilers and assemblers generate re-locatable object files
- Linkers generate executable object files

Executable and Linking Format (ELF)

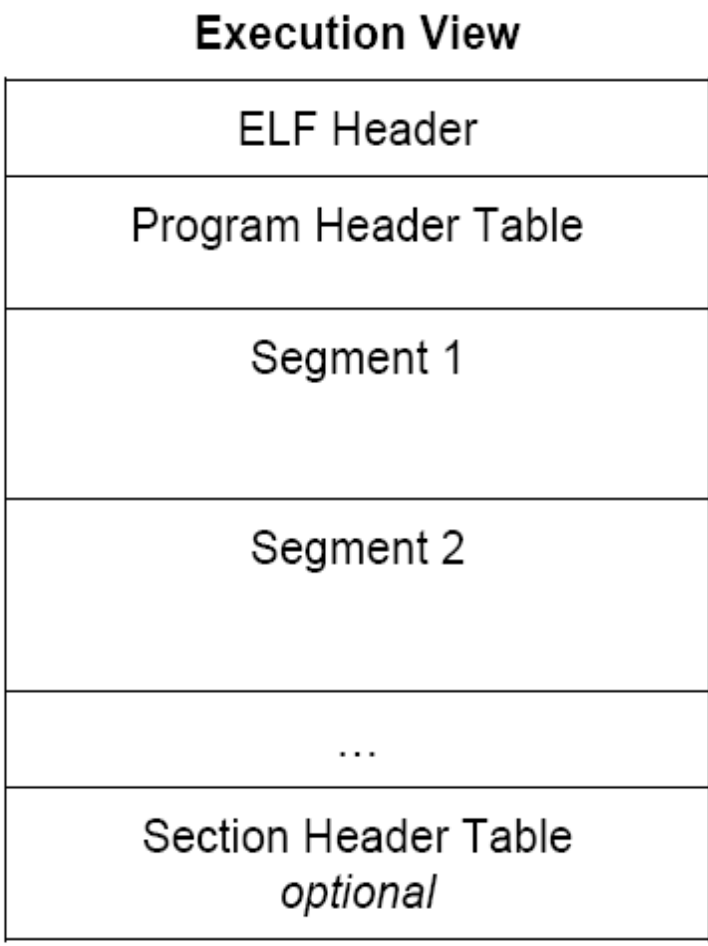
- Object files need to be in a specific format to facilitate linking and loading
- Executable and Linkable Format (ELF) is the popular format of an object file
- Supported by many vendors and tools
  - Diverse processors, multiple data encodings and multiple classes of machines
- ELF specifies the layout of the object files and not the contents of code or data
- ELF object files consist of
- ELF Header
  - Beginning of ELF file
  - Holds a roadmap of file's organization
  - How to interpret the file, independent of the processor
- Program header table
  - Tells the system how to create a process image
  - Files used to build a process image (execute a program) must have a program header table
  - Re-locatable files do not need one
- Sections
  - Object file information for the linking view
  - Instructions, data, symbol table, relocation information, etc.

Linking & Execution Views

Linking View
ELF Header
Program Header Table *optional*
Section 1
. . .
Section n
. . .
. . .
Section Header Table

Execution View
ELF Header
Program Header Table
Segment 1
Segment 2
. . .
Section Header Table *optional*

ELF Execution View




```
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half e_type;
    Elf32_Half e_machine;
    Elf32_Word e_version;
    Elf32_Addr e_entry;
    Elf32_Off  e_phoff;
    Elf32_Off  e_shoff;
    Elf32_Word e_flags;
    Elf32_Half e_ehsize;
    Elf32_Half e_phentsize;
    Elf32_Half e_phnum;
    Elf32_Half e_shentsize;
    Elf32_Half e_shnum;
    Elf32_Half e_shstrndx;
} Elf32_Ehdr;
```

ELF Header

- All ELF files contain a header in the beginning of the file
  - Determines whether the file is an ELF file, whether it is in big/little endian format, the target processor, offsets to the program header table and/or section header table…
- Format of the ELF header

```
#define EI_NIDENT 16
typedef struct {
    unsigned char e_ident[EI_NIDENT];  // file info (object file or not)
    Elf32_Half e_type;       // type of file (relocatable, executable, etc.)
    Elf32_Half e_machine;    // target processor (Intel x86, ARM, SPARC etc.)
    Elf32_Word e_version;    /  version # (to allow for future versions of ELF)
    Elf32_Addr e_entry;      // program entry point (0 if no entry point)
    Elf32_Off  e_phoff;      // offset of program header (in bytes)
    Elf32_Off  e_shoff;      // offset of section header table
    Elf32_Word e_flags;      // processor-specific flags
    Elf32_Half e_ehsize;     // ELF header's size
    Elf32_Half e_phentsize; // entry size in pgm header tbl
    Elf32_Half e_phnum;      // # of entries in pgm header
    Elf32_Half e_shentsize; // entry size in sec header tbl
    Elf32_Half e_shnum;      // # of entries in sec header tbl
    Elf32_Half e_shstrndx;  // sec header tbl index of str tbl
} Elf32_Ehdr;
```

See Section 3.2 of ARM ELF Specification document

ELF Sections

Relocatable files must have a section header table

Locations and size of sections are described by the section header table

	ELF header
Sections	`.text`
	`.rodata`
	`.data`
	`.bss`
	`.symtab`
	`.rel.text`
	`.rel.data`
	`.debug`
	`.line`
	`.strtab`
	Section header table

Source: "Computer Systems: A Programmer's Perspective", R. E. Bryant and D. O'Hallaron

Description of Various Sections

- `.text`: program instructions and literal data
- `.rodata`: Read-only data such as the format strings in printf statements
- `.data`: initialized global data
- `.bss`: un-initialized global data (set to zero when program image is created)
  - This section does not occupy any space in the object file
- `.symtab`: this section holds the symbol table information
  - All global variables and functions that are defined and referenced in the program
- `.rel.text`: list of locations in the `.text` section that will need to be modified when linker combines this object files with others
- `.rel.data`: relocation information for any global variables that are referenced or defined in a module
- `.debug`: debugging information (present only if code is compiled to produce debug information)
- `.line`: mapping between line numbers in C program and machine code instructions (present only if code is compiled to produce debug information)
- `.strtab`: string table for symbols defined in `.symtab` and `.debug` sections

Executable Object Files

0

Maps file sections to runtime memory segments

ELF header
Segment header table
`.init`
`.text`
`.rodata`
`.data`
`.bss`
`.symtab`
`.debug`
`.line`
`.strtab`
Section header table

Describes object file sections

Read-only memory segment (code segment)

Read/write memory segment (data segment)

Symbol table and debugging info are not loaded into memory

ELF Program Header

- Executable ELF files must have a program header table
  - The program header table is used to load the program (called "creating program image")
  - Each segment has its own entry in the program header table
    - e_*phnum* in ELF Header holds the number of program header entries
  - All program header entries have the same size (*e_phentsize* in ELF header)
- Program header entry for each segment

```
typedef struct {
    Elf32_Word p_type; //type of segment - loadable, dll,...
    Elf32_Off  p_offset;   //offset in bytes from the start of file
    Elf32_Addr p_vaddr;    //virtual address in memory of segment
    Elf32_Addr p_paddr;    //physical address in memory of segment
    Elf32_Word p_filesz;   //number of bytes in the file of the segment
    Elf32_Word p_memsz;    //number of bytes in memory of the process
          //image of the segment
    Elf32_Word p_flags;    //indicates whether segment is executable
    Elf32_Word p_align;    //alignment information
} Elf32_Phdr;
```

- What happens if `p_memsz > p_filesz`?
  - The remaining bytes `(p_memsz-p_filesz)` are initialized with 0

Useful Tools

- You can use many command line tools to parse ELF files
- ARM provides `readelf` command line utility that can display information about an ELF file
  - You can disassemble ELF files, look at symbol table information, etc.
- Example: `readelf main.o`

```
** ELF Header Information
File Name: main.c
   Machine class: ELFCLASS32 (32-bit)
   Data encoding: ELFDATA2MSB (Big endian)
   Header version: EV_CURRENT (Current version)
   File Type: ET_REL (Relocatable object) (1)
   Machine: EM_ARM (ARM)
   Header size: 52 bytes (0x34)
   Program header entry size: 32 bytes (0x20)
   Section header entry size: 40 bytes (0x28)
   Program header entries: 0
   Section header entries: 25
   Program header offset: 0 (0x00000000)
   Section header offset: 4512 (0x000011a0)
```

…and more

Useful Tools (contd.)

- Look at the symbol table information in `main.o`
- Example: `nm main.o`

D (global, initialized, data)	counter
T (global text)	main
U (global undefined)	square
t (local, static, text)	sum
C (global, uninitialized)	tmp

- **You can also use other switches to print information on each segment, section, print relocation information …**