Efficient Programming on ARM

Shiao-Li Tsao

Outline

- ARM System Software
  - ARM Bootstrap code
  - ARM OS code
  - Bootloader case study based on Uboot
  - Embedded OS case study based on uC/OS-II
- ARM Application Software
  - C compiler options
  - C programming skills on ARM
  - Assemble programming skills on ARM

Something Before Program Development

Slides are based on Vijay Kumar B., “Embedded Programming with the GNU Toolchain”

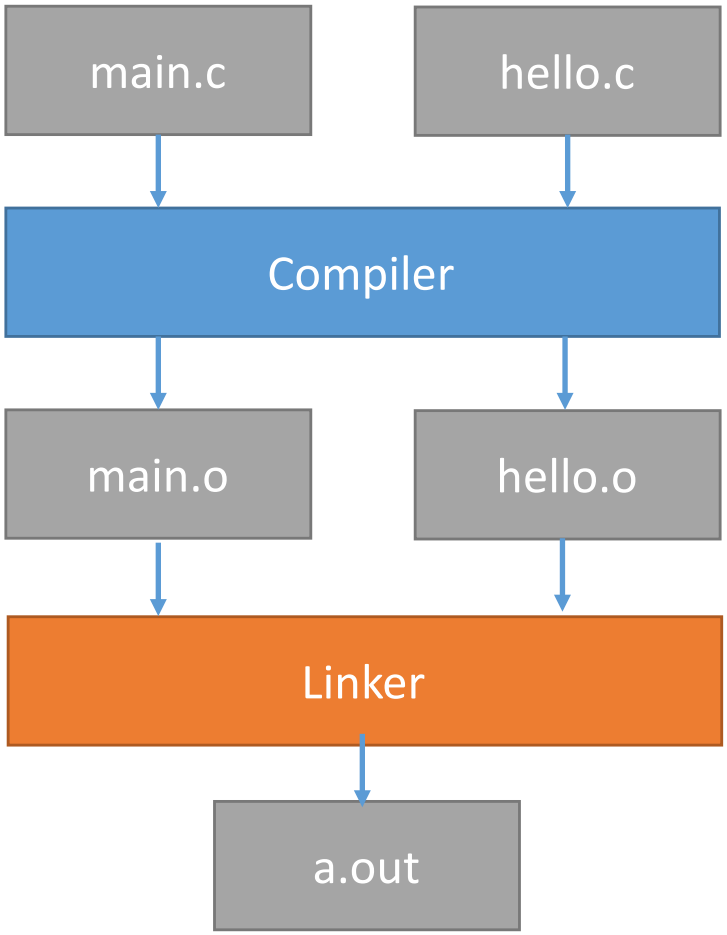
http://www.bravegnu.org/gnu-eprog/

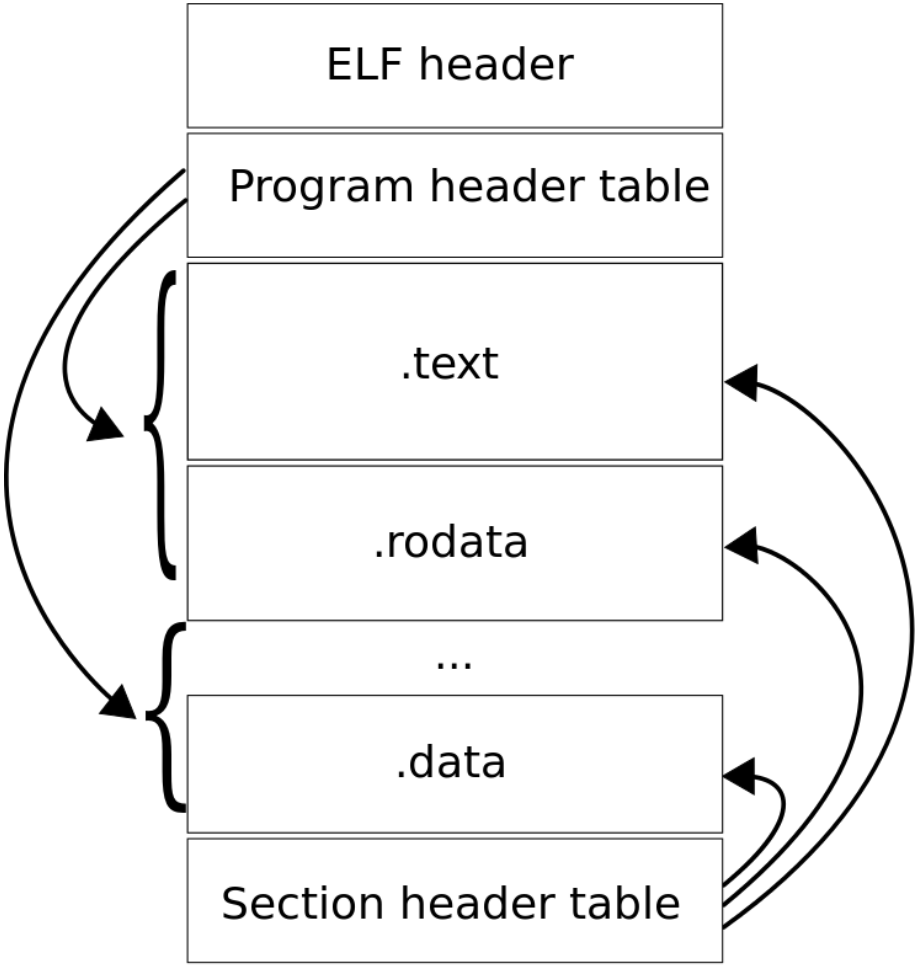
Linking

```c
// main.c
//
void sayHello();

char str[6] = {'h', 'e', 'l', 'l', 'o', 0};

int main(int argc, char *argv[])
{
  sayHello();
  return 0;
}
```

```c
// hello.c
//
#include <stdio.h>

extern char str[6];

void sayHello() {
  printf("%s\n", str);
}
```

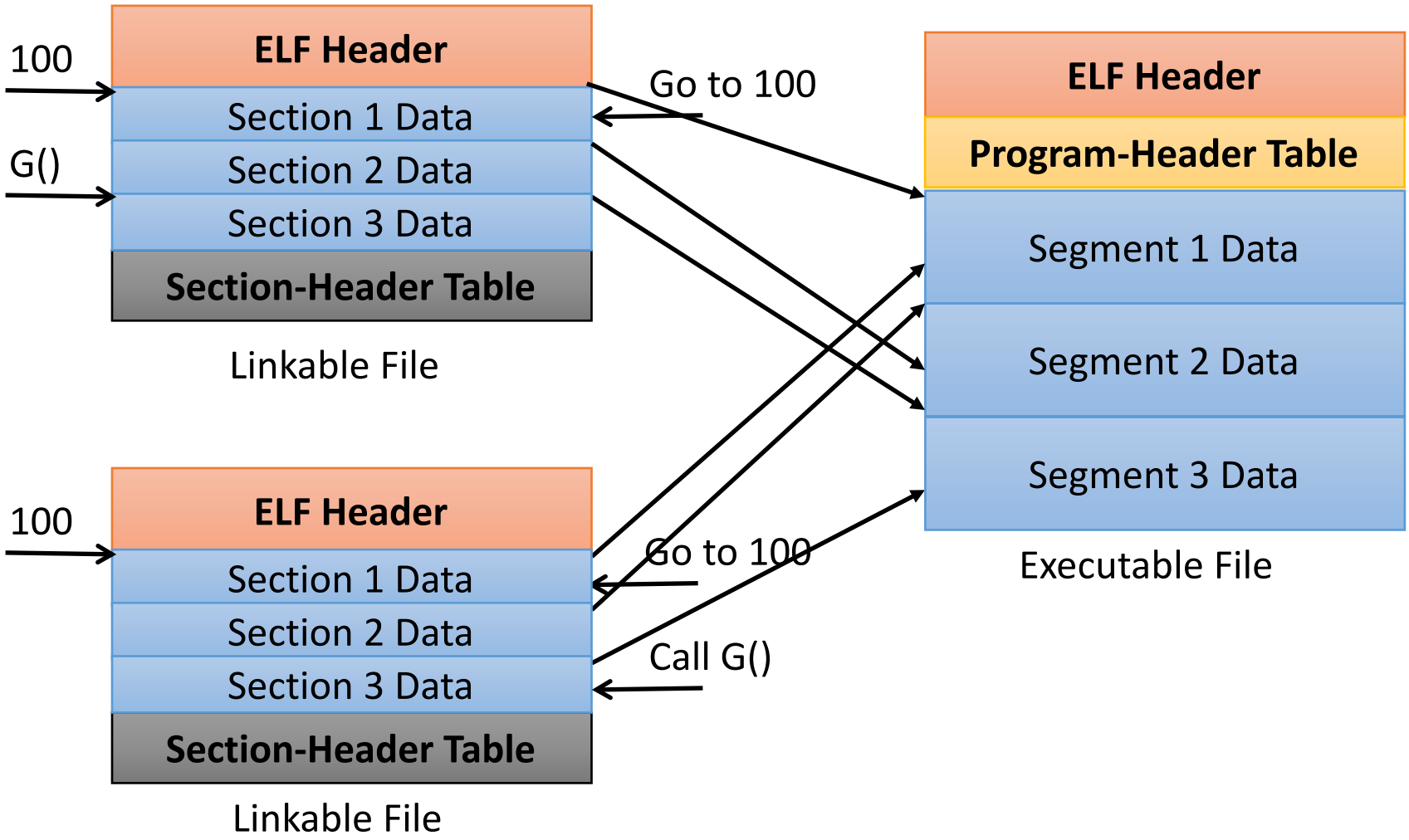
Linking





By Surueña - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2922583

Linking

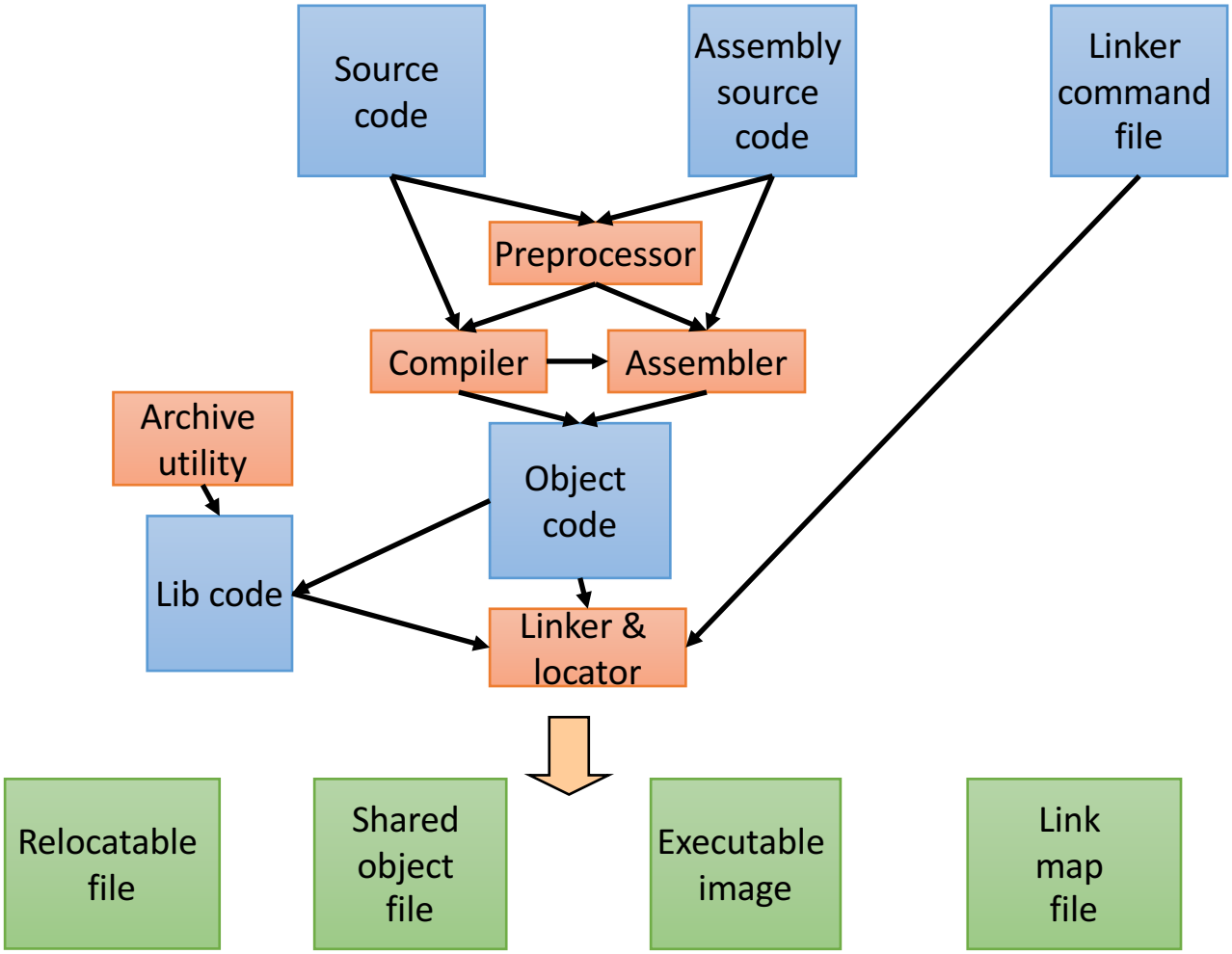

100
G()
ELF Header
Section 1 Data
Section 2 Data
Section 3 Data
Section-Header Table
Linkable File
100
ELF Header
Section 1 Data
Section 2 Data
Section 3 Data
Section-Header Table
Linkable File
Go to 100
Go to 100
Call G()
ELF Header
Program-Header Table
Segment 1 Data
Segment 2 Data
Segment 3 Data
Executable File

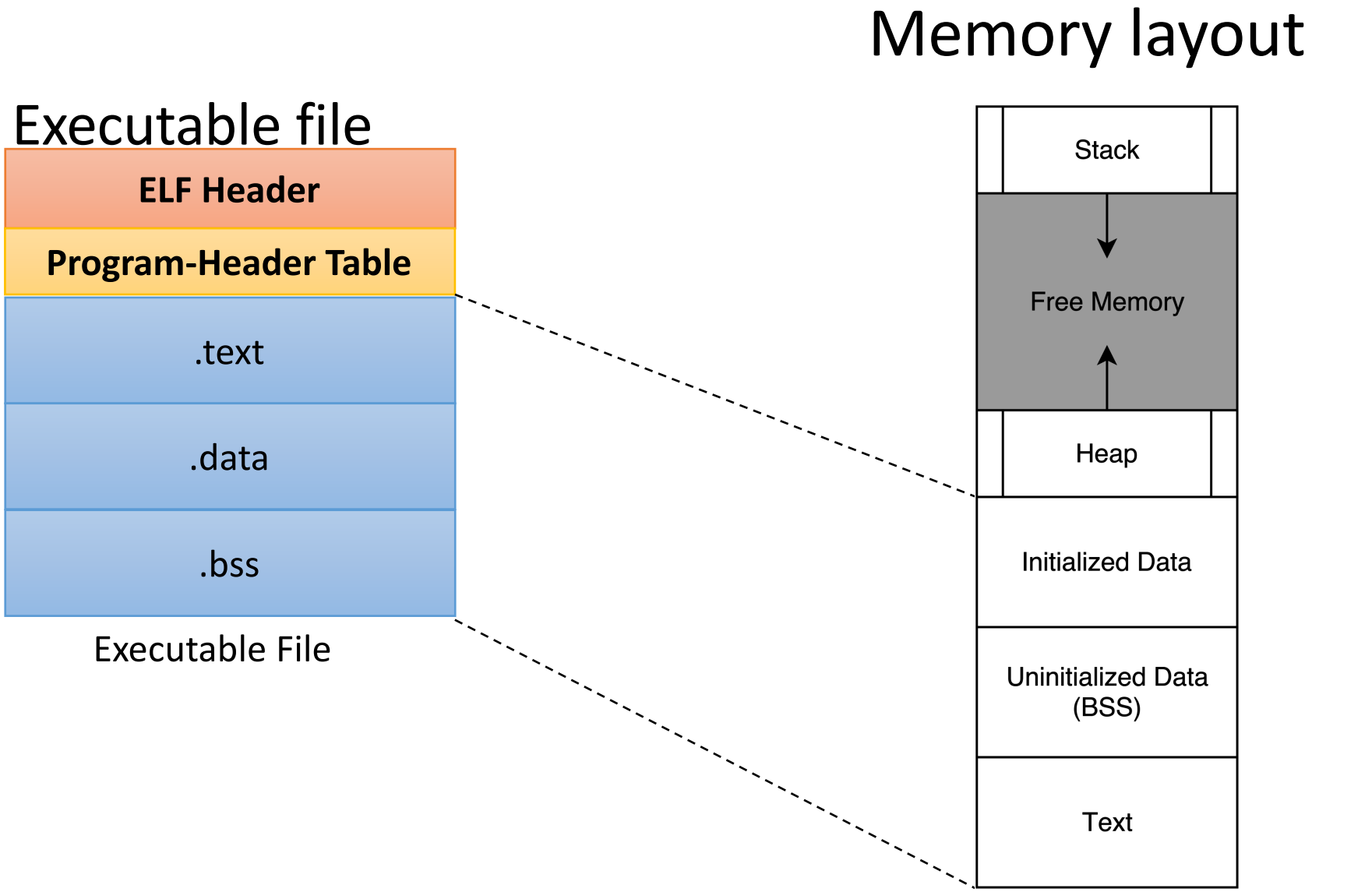
Linking

- Please review system programming and compiler if you are not familiar with below terms
  - Static linking
  - Dynamic linking
  - Relocations
  - Symbol table
  - Share library
  - Linking and loading

How to compile kernel codes

- Creating an executable image





Assembly, Machine code, Memory dump

Assembly ->

```
Dump of assembler code for function system_call:
   0x000076d8 <+0>:     cmp    $0x47,%eax
   0x000076db <+3>:     ja     0x76c8 <bad_sys_call>
   0x000076dd <+5>:     push   %ds
   0x000076de <+6>:     push   %es
   0x000076df <+7>:     push   %fs
   0x000076e1 <+9>:     push   %edx
=> 0x000076e2 <+10>:    push   %ecx
   0x000076e3 <+11>:    push   %ebx
   0x000076e4 <+12>:    mov    $0x10,%edx
   0x000076e9 <+17>:    mov    %edx,%ds
   0x000076eb <+19>:    mov    %edx,%es
   0x000076ed <+21>:    mov    $0x17,%edx
   0x000076f2 <+26>:    mov    %edx,%fs
   0x000076f4 <+28>:    call   *0x1a020(,%eax,4)
   0x000076fb <+35>:    push   %eax
   0x000076fc <+36>:    mov    0x1b140,%eax
   0x00007701 <+41>:    cmpl   $0x0,(%eax)
   0x00007704 <+44>:    jne    0x76ce <reschedule>
   0x00007706 <+46>:    cmpl   $0x0,0x4(%eax)
   0x0000770a <+50>:    je     0x76ce <reschedule>
End of assembler dump.
```

Memory dump

|

v

```
0x76d8 <system_call>:   131     248     71      119     235     30      6       15
0x76e0 <system_call+8>: 160     82      81      83      186     16      0       0
0x76e8 <system_call+16>:        0       142     218     142     194     186     23      0
0x76f0 <system_call+24>:        0       0       142     226     255     20      133     32
0x76f8 <system_call+32>:        160     1       0       80      161     64      177     1
0x7700 <system_call+40>:        0       131     56      0       117     200     131     120
0x7708 <system_call+48>:        4       0
```

Sections (Contd.)

```
        .data
arr:    .word 10, 20, 30, 40, 50
len:    .word 5
        .text
start:  mov r1, #10
        mov r2, #20
        .data
result: .skip 4
        .text
        add r3, r2, r1
        sub r3, r2, r1
```

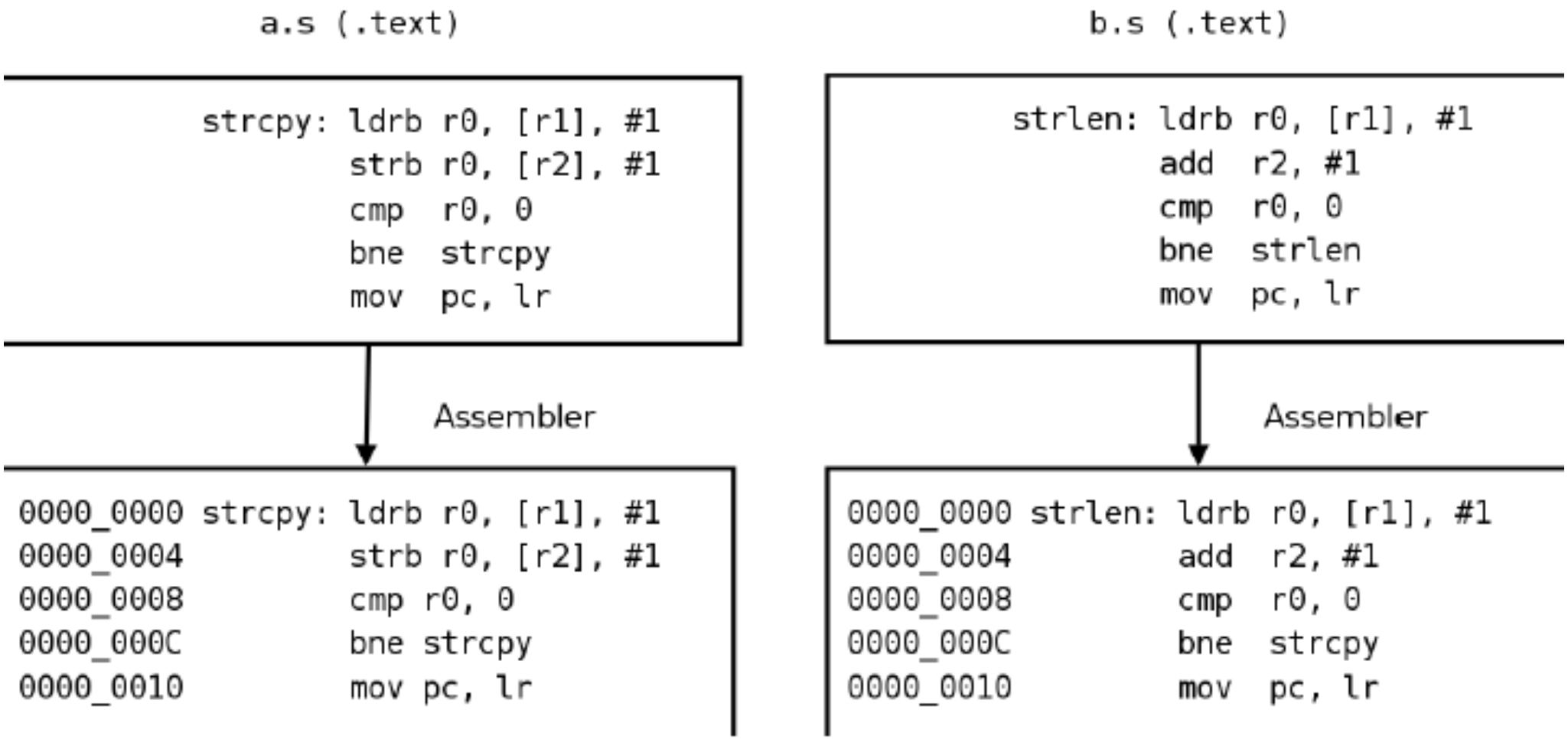
.data section

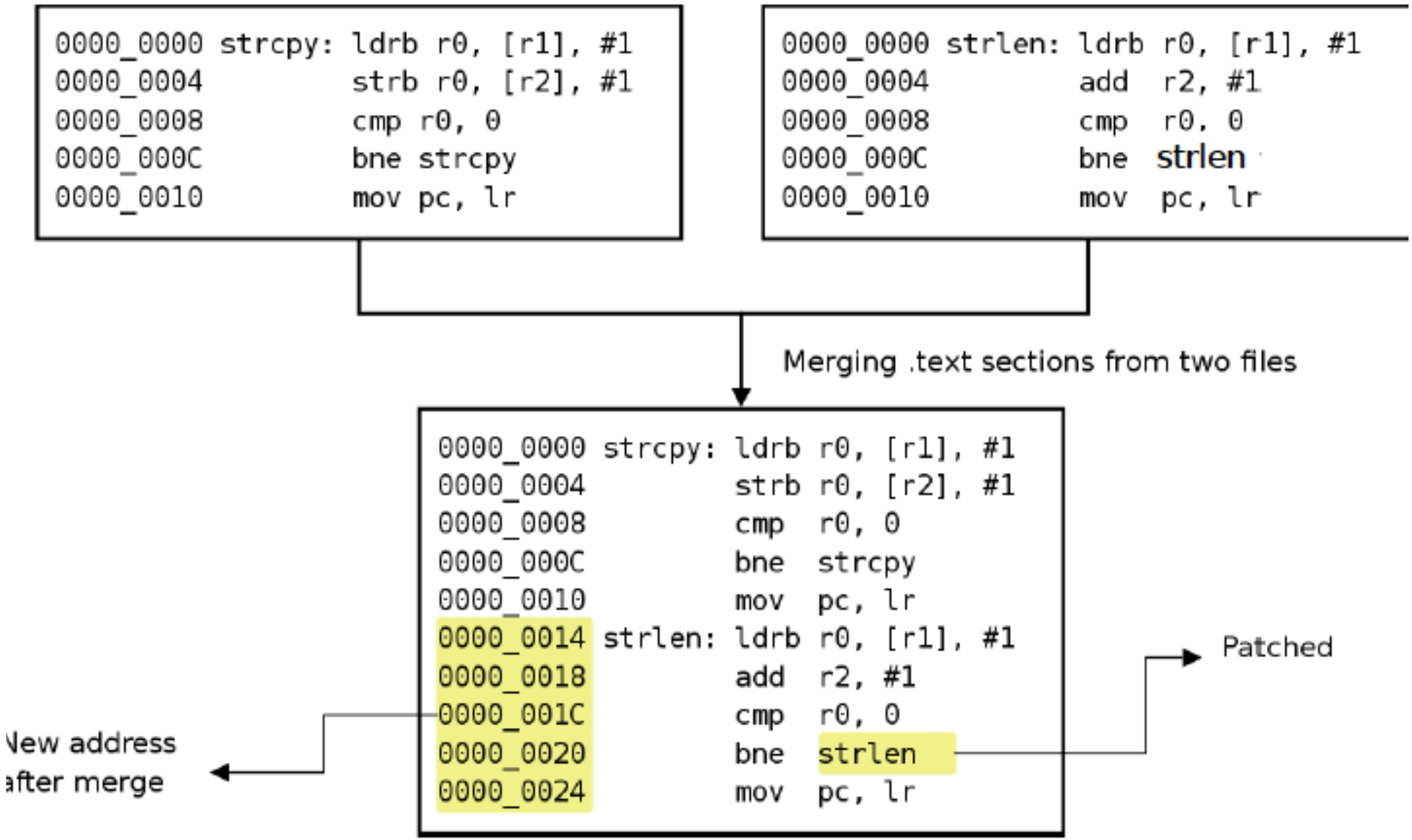
```
0000_0000 arr:    .word 10, 20, 30, 40, 50
0000_0014 len:    .word 5
0000_0018 result: .skip 4
```

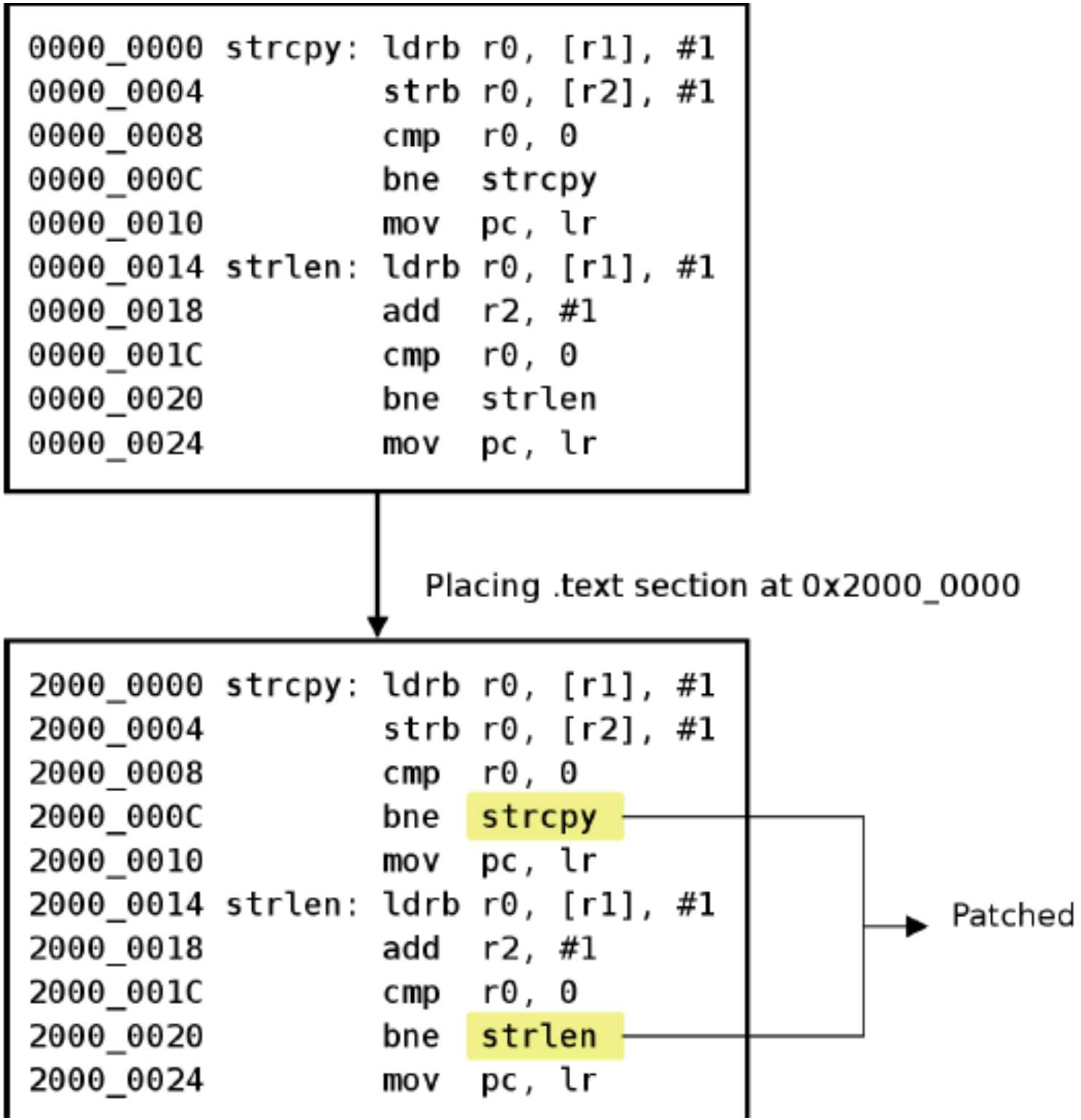
.text section

```
0000_0000 start:  mov r1, #10
0000_0004         mov r2, #20
0000_0008         add r3, r2, r1
0000_000C         sub r3, r2, r1
```

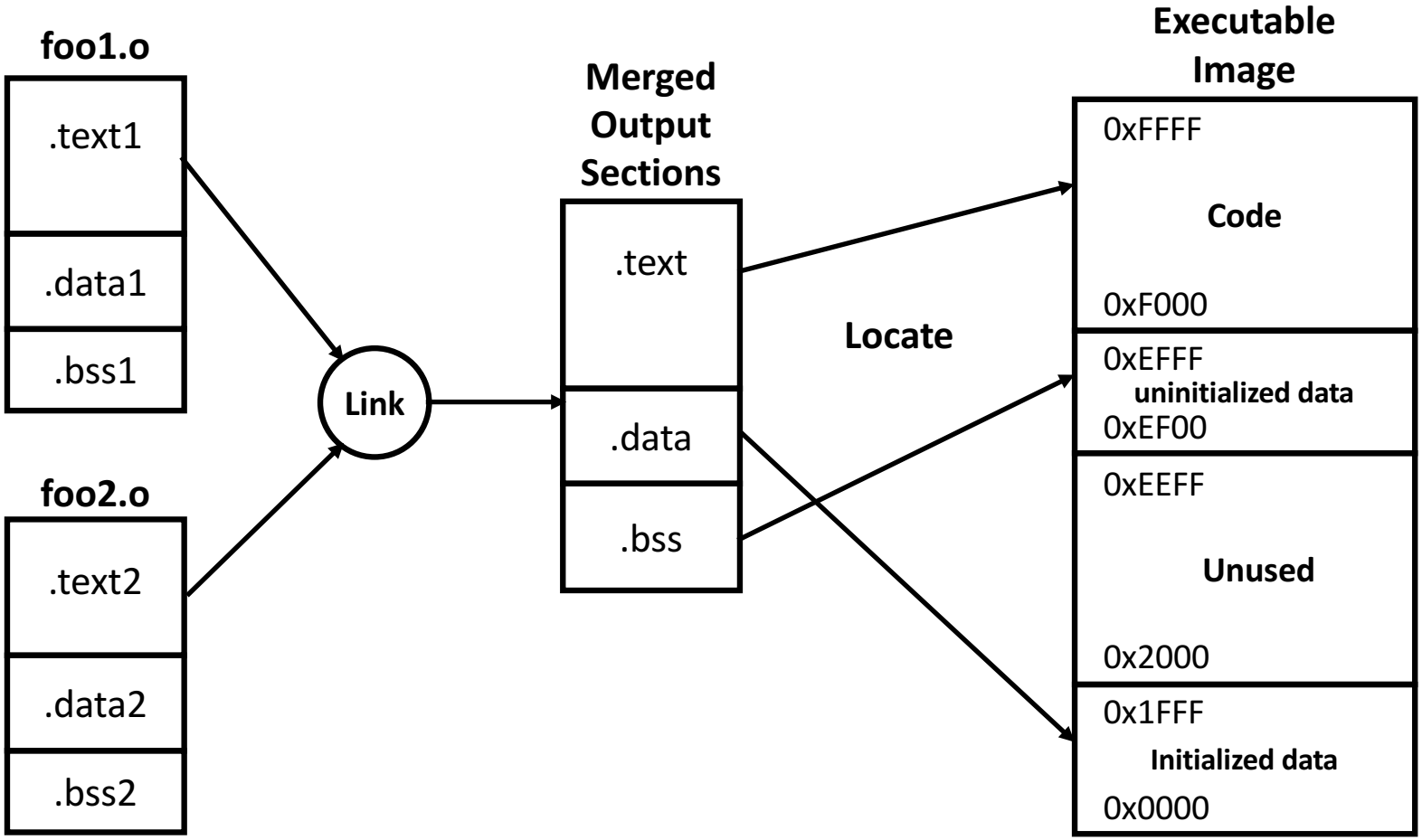
- Source – sections can be interleaved
- Bytes of a section – contiguous addresses


a.s (.text)
strcpy: ldrb r0, [r1], #1
strb r0, [r2], #1
cmp r0, 0
bne strcpy
mov pc, lr
Assembler
0000_0000 strcpy: ldrb r0, [r1], #1
0000_0004 strb r0, [r2], #1
0000_0008 cmp r0, 0
0000_000C bne strcpy
0000_0010 mov pc, lr
b.s (.text)
strlen: ldrb r0, [r1], #1
add r2, #1
cmp r0, 0
bne strlen
mov pc, lr
Assembler
0000_0000 strlen: ldrb r0, [r1], #1
0000_0004 add r2, #1
0000_0008 cmp r0, 0
0000_000C bne strcpy
0000_0010 mov pc, lr


0000_0000 strcpy: ldrb r0, [r1], #1
0000_0004 strb r0, [r2], #1
0000_0008 cmp r0, 0
0000_000C bne strcpy
0000_0010 mov pc, lr
0000_0000 strlen: ldrb r0, [r1], #1
0000_0004 add r2, #1
0000_0008 cmp r0, 0
0000_000C bne strlen
0000_0010 mov pc, lr
Merging .text sections from two files
0000_0000 strcpy: ldrb r0, [r1], #1
0000_0004 strb r0, [r2], #1
0000_0008 cmp r0, 0
0000_000C bne strcpy
0000_0010 mov pc, lr
0000_0014 strlen: ldrb r0, [r1], #1
0000_0018 add r2, #1
0000_001C cmp r0, 0
0000_0020 bne strlen
0000_0024 mov pc, lr
Patched
New address
after merge

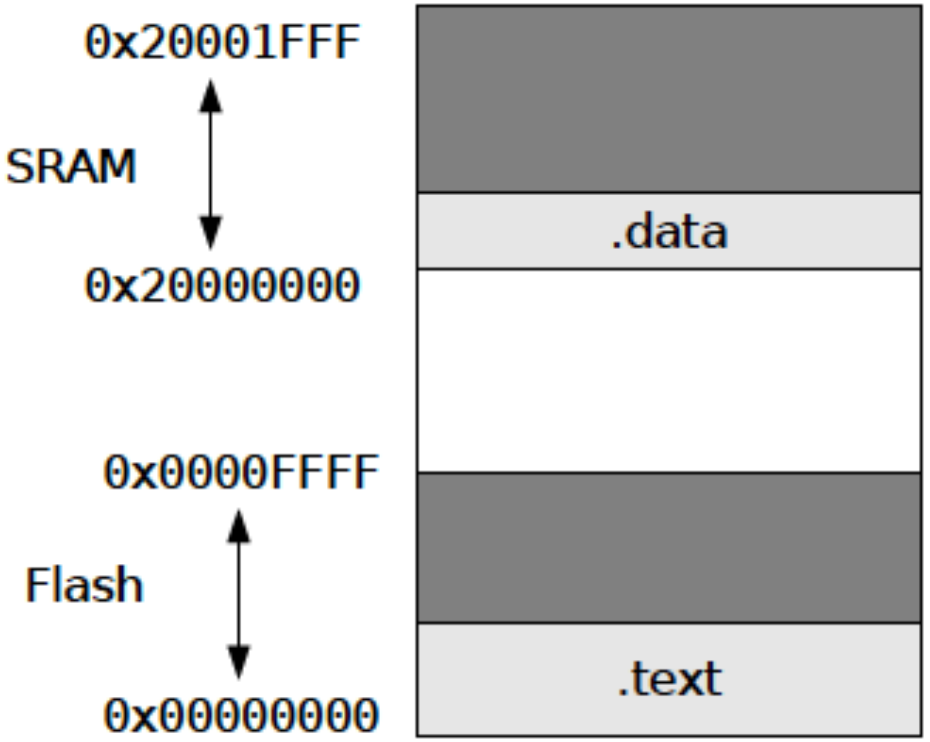

0000_0000 strcpy: ldrb r0, [r1], #1
0000_0004 strb r0, [r2], #1
0000_0008 cmp r0, 0
0000_000C bne strcpy
0000_0010 mov pc, lr
0000_0014 strlen: ldrb r0, [r1], #1
0000_0018 add r2, #1
0000_001C cmp r0, 0
0000_0020 bne strlen
0000_0024 mov pc, lr
Placing .text section at 0x2000_0000
2000_0000 strcpy: ldrb r0, [r1], #1
2000_0004 strb r0, [r2], #1
2000_0008 cmp r0, 0
2000_000C bne strcpy
2000_0010 mov pc, lr
2000_0014 strlen: ldrb r0, [r1], #1
2000_0018 add r2, #1
2000_001C cmp r0, 0
2000_0020 bne strlen
2000_0024 mov pc, lr
Patched

Linker and Locator Flows



Linker Script Revisited

```
MEMORY {
     FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x10000
     SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x2000
}

SECTIONS {
     .text : {
          * (.text);
     } > FLASH

     .data : {
          * (.data);
     } > SRAM
}
```



Make and Makefile

- Make: utility to provide a convenient facility to build, install, and uninstall projects
- Makefile: script file for make to compile and link programs

Make and Makefile

```
target ... : prerequisites ...
        recipe
        ...
        ...
```

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

http://www.gnu.org/software/make/manual/make.pdf

Make and Makefile

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        rm edit $(objects)
```

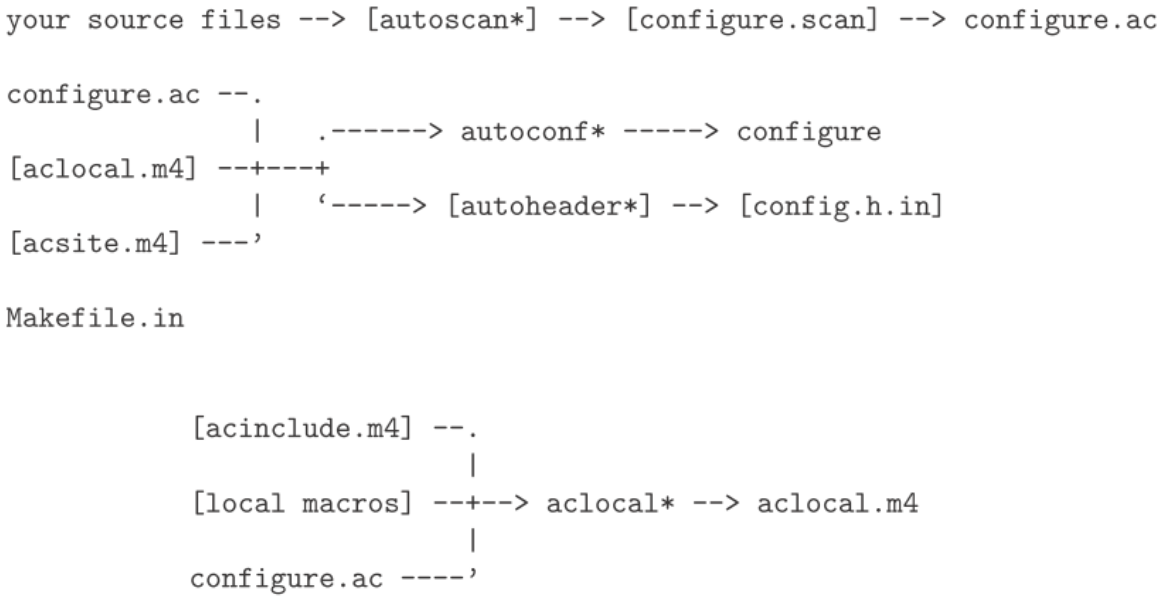
http://www.gnu.org/software/make/manual/make.pdf

Make and Makefile

- Variables and settings
  - make config
  - make menuconfig
- Phony targets
  - make all
  - make clean
  - make depend
  - make install
  - make uninstall

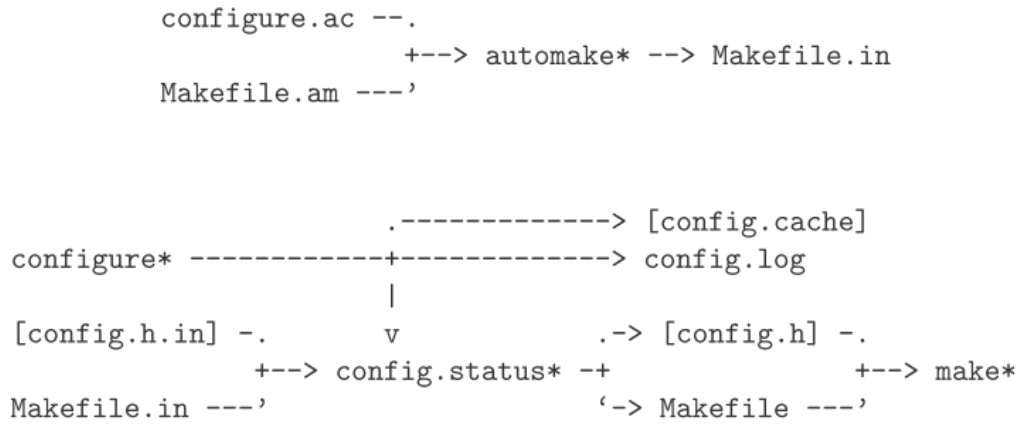
The portability problem

- Hardware differences
- OS differences
- Compiler differences



Autoconf Manual - The GNU Operating System

The portability problem



Autoconf Manual - The GNU Operating System

Minimal Bootloader

Minimal Bootloader Procedures

Step	Description
1	Take the Reset exception
2	Start initializing the hardware
3	Remap memory
4	Initialize communication hardware
5	Bootloader—copy payload and relinquish control

Step 1: Take the Reset Exception

```
        AREA start,CODE,READONLY
        ENTRY

sandstone_start
        B       sandstone_init1       ; reset vector
        B       ex_und                ; undefined vector
        B       ex_swi                ; swi vector
        B       ex_pabt               ; prefetch abort vector
        B       ex_dabt               ; data abort vector
        NOP                           ; not used...
        B       int_irq               ; irq vector
        B       int_fiq               ; fiq vector

ex_und  B       ex_und                ; loop forever
ex_swi  B       ex_swi                ; loop forever
ex_dabt B       ex_dabt               ; loop forever
ex_pabt B       ex_pabt               ; loop forever
int_irq B       int_irq               ; loop forever
int_fiq B       int_fiq               ; loop forever
```

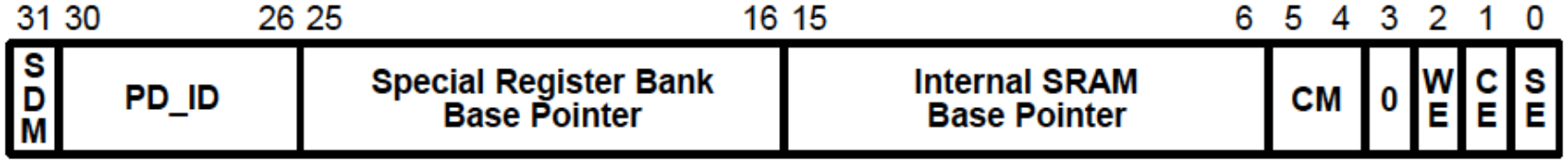
Step 1: Take the Reset Exception

- Dummy handlers are set up
- Control is passed to code to initialize the hardware

Step 2: Start Initializing the Hardware

```
sandstone_init1
        LDR     r3, =SYSCFG         ; where SYSCFG=0x03ff0000
        LDR     r4, =0x03ffffa0
        STR     r4, [r3]
```

Address range	Size	Description
0x00000000 to 0x0003FFFF	256KB	32 bit SRAM bank, using ROMCON1
0x00040000 to 0x0007FFFF	256KB	32 bit SRAM bank, using ROMCON2
0x01800000 to 0x0187FFFF	512KB	16 bit flash bank, using ROMCON0
0x03FE0000 to 0x03FE1FFF	8KB	32 bit internal SRAM
0x03FF0000 to 0x03FFFFFF	64KB	Microcontroller register space



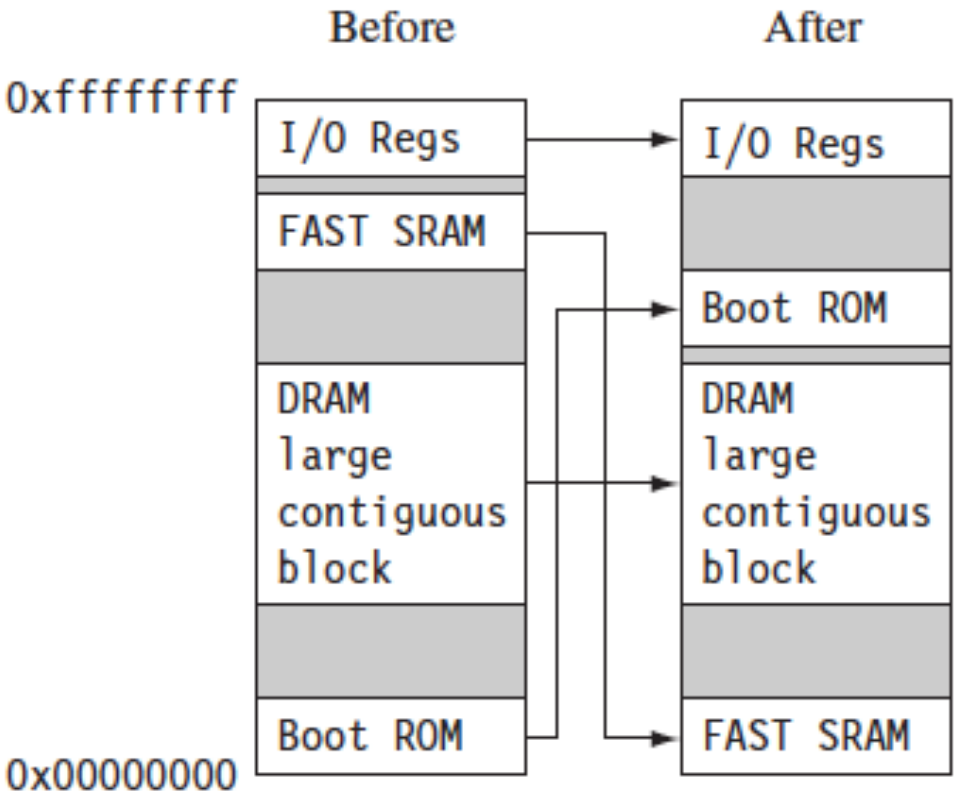
Step 2: Start Initializing the Hardware

```
; -- Section 2 : Initialize the hardware to use the segment
display.

        LDR     r2, =SEG_MASK
        LDR     r3, =IOPMOD
        LDR     r4, [r3]
        ORR     r4, r4, r2
        STR     r4, [r3]

        LDR     r3, =IOPDATA
        LDR     r4, [r3]
        ORR     r4, r4, r2
        STR     r4, [r3]

        LDR     r2,=nSEG_MASK

; -- To show hardware is configured correctly display 0 on
; -- the segment display.

        LDR     r5,data_dis0
        BL      _e7t_setSegmentDisplay
```

Step 2: Start Initializing the Hardware

- The system registers are set from a known base address—0x03ff0000
- The segment display is configured, so that it can be used to display progress

Step 3: Remap Memory



Step 3: Remap Memory

Memory type	Start address	End address	Size
Flash ROM	0x00000000	0x00080000	512K
SRAM bank 0	Unavailable	unavailable	256K
SRAM bank 1	Unavailable	unavailable	256K

```
        LDR      lr, =sandstone_init2
        LDR      r4, =0x1800000
        ADD      lr,lr,r4

; -- Load in the target values into the
control registers

        ADRL     r0, memorymaptable_str
        LDMIA    r0, {r1-r12}
        LDR      r0, =EXTDBWTH

; -- remap and jump to ROM code

        STMIA    r0, {r1-r12}
        MOV      pc, lr
```

Step 3: Remap Memory

```
memorymaptable_str
        DCD rEXTDBWTH     ; ROM0(Half), ROM1(Word), ROM1(Word), rest Disabled
        DCD rROMCON0      ; 0x1800000 ~ 0x1880000, RCS0, 4Mbit, 4cycle
        DCD rROMCON1      ; 0x0000000 ~ 0x0040000, RCS1, 256KB, 2cycle
        DCD rROMCON2      ; 0x0040000 ~ 0x0080000, RCS2, 256KB, 2cycle
```

```
; -- ROMCON0 : ROM Bank0 Control register ..............................

ROMBasePtr0     EQU  0x180:SHL:10   ;=0x1800000
ROMEndPtr0      EQU  0x188:SHL:20   ;=0x1880000
PMC0            EQU  0x0            ; 0x0=Normal ROM, 0x1=4Word Page
                                    ; 0x2=8Word Page, 0x3=16Word Page
rTpa0           EQU  (0x0:SHL:2)    ; 0x0=5Cycle, 0x1=2Cycle
                                    ; 0x2=3Cycle, 0x3=4Cycle
rTacc0          EQU  (0x3:SHL:4)    ; 0x0=Disable, 0x1=2Cycle
                                    ; 0x2=3Cycle, 0x3=4Cycle
                                    ; 0x4=5Cycle, 0x5=6Cycle
                                    ; 0x6=7Cycle, 0x7=Reserved
rROMCON0   EQU  ROMEndPtr0+ROMBasePtr0+rTacc0+rTpa0+PMC0
```

Step 3: Remap Memory

Type	Start address	End address	Size
Flash ROM	0x01800000	0x01880000	512K
SRAM bank 0	0x00000000	0x00040000	256K
SRAM bank 1	0x00040000	0x00080000	256K

- Memory has been remapped
- pc now points to the next step. This address is located in the newly remapped flash ROM

Step 4: Initialize Communication Hardware

```
sandstone_init2

        LDR     r5,data_dis4
        BL      _e7t_setSegmentDisplay

        B       sandstone_serialcode
```

- Serial port initialized—9600 baud, no parity, one stop bit, and no flow control
- Sandstone banner sent out through the serial port

Step 5: Bootloader – Copy Payload and Relinquish Control

```
sandstone_load_and_boot
; -- Section 1 : Copy payload to address 0x00000000
        MOV     r13,#0
        LDR     r12,payload_start_address
        LDR     r14,payload_end_address
block_copy
        LDMIA   r12!,{r0-r11}
        STMIA   r13!,{r0-r11}
        CMP     r12,r14
        BLE     block_copy
; -- Section 2 : Relingish control over to the payload
        MOV     pc,#0
    LTORG
; -- Data Section : Payload information inserted at build time.
payload_start_address
        DCD     startAddress
payload_end_address
        DCD     endAddress
```

Step 5: Bootloader – Copy Payload and Relinquish Control

- Payload copied into SRAM, address 0x00000000
- Control of the pc is relinquished to the payload; pc = 0x00000000
- The system is completely booted

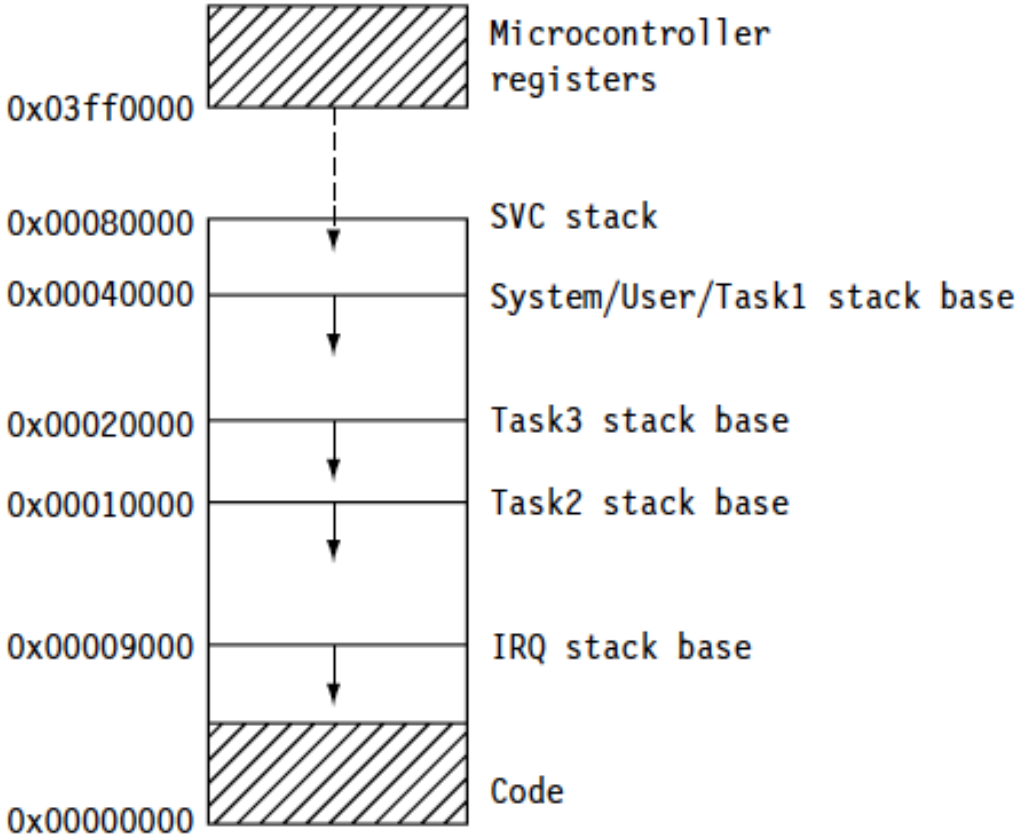
Minimal Operating System

Fundamental Components

- Initialization
- Memory handling (static memory management)
- Interrupt/exception handling
- Scheduler and context switch
- Device driver framework

Memory Map

- Non-MMU and Non-MPU



Step 1: Initiation

```
 LDR     pc,vectorReset
     LDR     pc,vectorUndefined
     LDR     pc,vectorSWI
     LDR     pc,vectorPrefetchAbort
     LDR     pc,vectorDataAbort
     LDR     pc,vectorReserved
     LDR     pc,vectorIRQ
     LDR     pc,vectorFIQ

; -- Kernel Jump table ------------------------------
vectorReset         DCD     coreInitialize
vectorUndefined     DCD     coreUndefinedHandler
vectorSWI           DCD     coreSWIHandler
vectorPrefetchAbort DCD     corePrefetchAbortHandler
vectorDataAbort     DCD     coreDataAbortHandler
vectorReserved      DCD     coreReservedHandler
vectorIRQ           DCD     coreIRQHandler
vectorFIQ           DCD     coreFIQHandler

     END
```

Step 1: Initiation (Cont.)

```
coreInitialize
; ------------------------------------------------
; Setup stacks for SVC,IRQ,SYS&USER
; ------------------------------------------------
     MOV     sp,#0x80000           ; initializing sp to TopOfMemory
     MSR     CPSR_c,#NoInt|SYS32md
     MOV     sp,#0x40000           ; SYS/User Stack = 0x40000
     MSR     CPSR_c,#NoInt|IRQ32md
     MOV     sp,#0x9000            ; IRQ Stack = 0x9000
     MSR     CPSR_c,#NoInt|SVC32md
; ------------------------------------------------
; Setup Task Process Control Block (PCB).
; ------------------------------------------------
     LDR     r0,=C_EntryTask2     ; addr of C_EntryTask2
     LDR     r1,=PCB_PtrTask2
     MOV     r2,#0x10000
     BL      pcbSetUp
; -- set the current ID to TASK1 ..........
     LDR     r0, =PCB_CurrentTask
     MOV     r1, #0
     STR     r1,[r0]               ; first task ID = 0
     LDR     lr,=C_Entry
     MOV     pc,lr                 ; call C_Entry
     END
```

Step 1: Initiation (Cont.)

```
int C_Entry(void)
{
lltrace(cinit_init(),CINITINIT);
lltrace(eventTickStart(),TICKSTART);
STATE=BOOT_SLOS;
  __asm
  {
  MSR             CPSR_c,#0x50
  }
lltrace(C_EntryTask1(),ENTERTASK);
return 0;
}
```

Step 1: Initiation (Cont.)

```
void C_EntryTask1(void)
{
volatile int delay;
  if (openRedLED ()==0)
  {
    while (1)
    {
    aWAIT;
    switchOnRedLED ();
    /* dummy time delay... */
    for (delay=0; delay<0x20ffff; delay++) {}
    aSIGNAL;
    switchOffRedLED ();
    /* dummy time delay... */
    for (delay=0; delay<0x20ffff; delay++) {}
    }
  }
while (1)
  {
  delay=0xBEEFBEEF;
  };
}
```

Step 3: Interrupt and Exception Handling

Exception	Purpose
Reset	initialize the operating system
SWI	mechanism to access device drivers
IRQ	mechanism to service events

Step 3: Interrupt and Exception Handling (Cont.)

```
coreSWIHandler
        STMFD   sp!,{r0-r12,r14}     ; save context
        LDR     r10,[r14,#-4]        ; load SWI instruction
        BIC     r10,r10,#0xff000000 ; mask off the MSB 8 bits
        MOV     r1,r13               ; copy r13_svc to r1
        MRS     r2,spsr              ; copy spsr to r2
        STMFD   r13!,{r2}            ; save r2 onto the stack
        BL      swi_jumptable        ; branch to the swi_jumptable

        LDMFD   r13!,{r2}           ; restore the r2 (spsr)
        MSR     spsr_cxsf,r2        ; copy r2 back to spsr
        LDMFD   r13!,{r0-r12,pc}^   ; restore context and return
swi_jumptable
        MOV     r0,r10              ; move the SWI number to r0
        B       eventsSWIHandler    ; branch to SWI handler
```

```
void eventsSWIHandler(int swi_number, SwiRegs *r)
{
  if (swi_number==SLOS)
  {
    if (r->r[0]==Event_IODeviceInit)
    {
    /* do not enable IRQ interrupts .... */
    io_initialize_drivers ();
    }
    else
    {
      /* if not initializing change to system mode
         and enable IRQs */
      if (STATE!=1) {modifyControlCPSR (SYSTEM|IRQoN);}

      switch (r->r[0])
      {
      case /* SWI */ Event_IODeviceOpen:
        r->r[0] =
          (unsigned int) io_open_driver
          (
          /*int *ID */ (UID *)r->r[1],
          /*unsigned major_device */ r->r[2],
          /*unsigned minor_device */ r->r[3]
          );
```

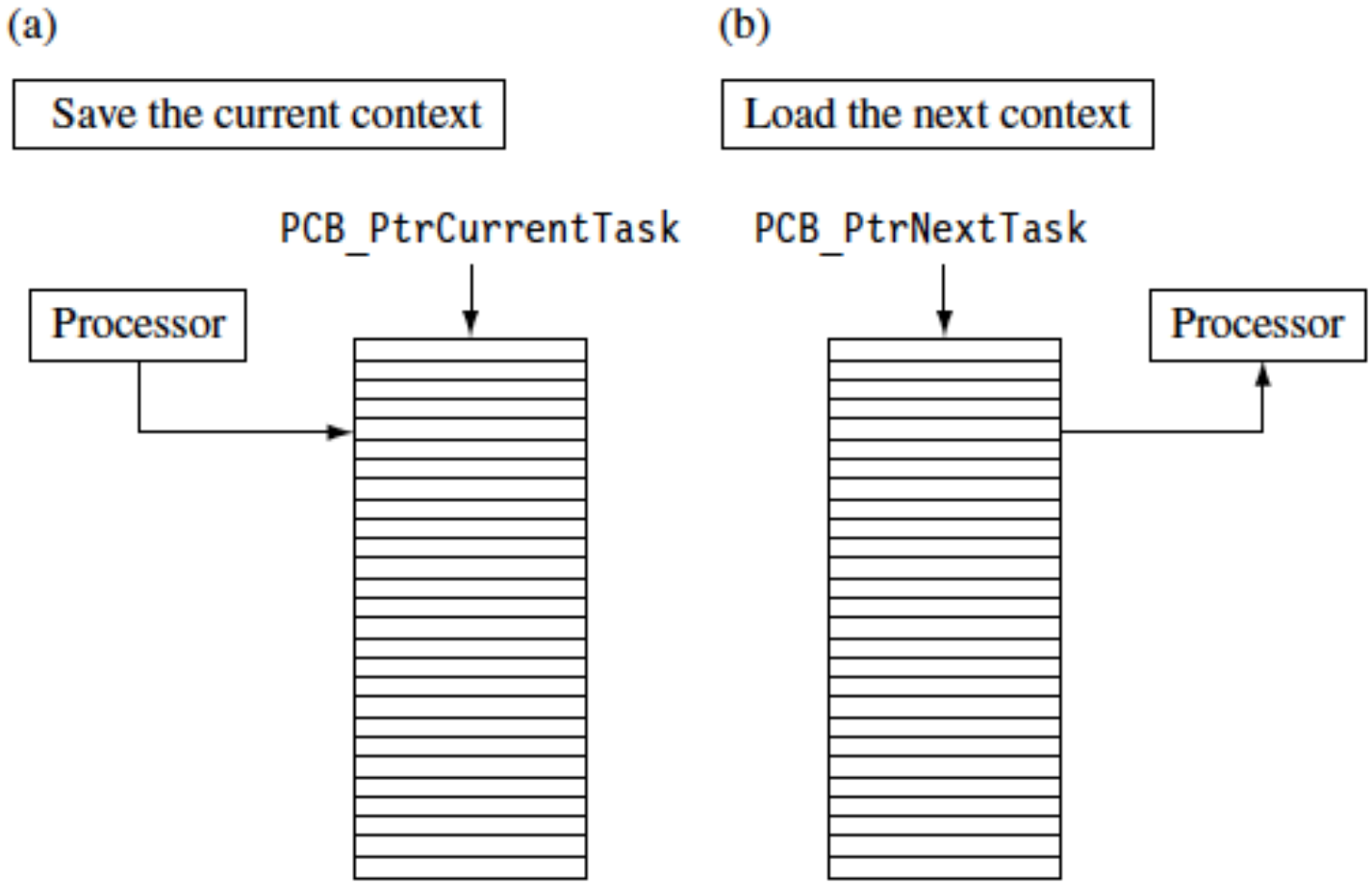
Step 3: Interrupt and Exception Handling (Cont.)

```
TICKINT   EQU 0x400
BUTTONINT EQU 0x001

eventsIRQHandler
        SUB     r14, r14, #4                ; r14_irq-=4
        STMFD   r13!, {r0-r3, r12, r14}     ; save context
        LDR     r0,INTPND                   ; r0=int pending reg
        LDR     r0,[r0]                     ; r0=memory[r0]
        TST     r0,#TICKINT                 ; if tick int
        BNE     eventsTickVeneer            ;   then tick ISR
        TST     r0,#BUTTONINT               ; if button interrupt
        BNE     eventsButtonVeneer          ;   then button ISR
        LDMFD   r13!, {r0-r3, r12, pc}^     ; return to task

eventsTickVeneer
        BL      eventsTickService           ; reset tick hardware
        B       kernelScheduler             ; branch to scheduler
```

Step 4: Scheduler and Context Switch

```
task t=0,t';

scheduler()
{
  t' = t + 1;
  if t' = MAX_NUMBER_OF_TASKS then
    t' = 0 // the first task.
  end;

  ContextSwitch(t,t')
}
```

Step 4: Scheduler and Context Switch (Cont.)



Step 4: Scheduler and Context Switch (Cont.)

- Saving registers to a PCB

```
Offset15Regs EQU 15*4

handler_contextswitch
        LDMFD   r13!,{r0-r3,r12,r14}    ; [1.1] restore registers
        LDR     r13,=PCB_PtrCurrentTask ; [1.2]
        LDR     r13,[r13]               ; r13=mem32[r13]
        SUB     r13,r13,#Offset15Regs   ; r13-=15*Reg:place r13
        STMIA   r13,{r0-r14}^           ; [1.3] save user mode registers
        MRS     r0, spsr                ; copy spsr
        STMDB   r13,{r0,r14}            ; save r0(spsr) & r14(lr)
```

Step 4: Scheduler and Context Switch (Cont.)

- Loading registers from a PCB

```
LDR     r13,=PCB_PtrNextTask  ; [2.1] r13=PCB_PtrNextTask
LDR     r13,[r13]             ; r13=mem32[r13] : next PCB
SUB     r13,r13,#Offset15Regs ; r13-=15*Registers
LDMDB   r13,{r0,r14}          ; [2.2] load r0 & r14
MSR     spsr_cxsf, r0         ; spsr = r0
LDMIA   r13,{r0-r14}^         ; load r0_user-r14_user
LDR     r13,=PCB_IRQStack     ; [2.3] r13=IRQ stack addr
LDR     r13,[r13]             ; r13=mem32[r13] : reset IRQ
MOVS    pc,r14                ; [2.4] return to next task
```

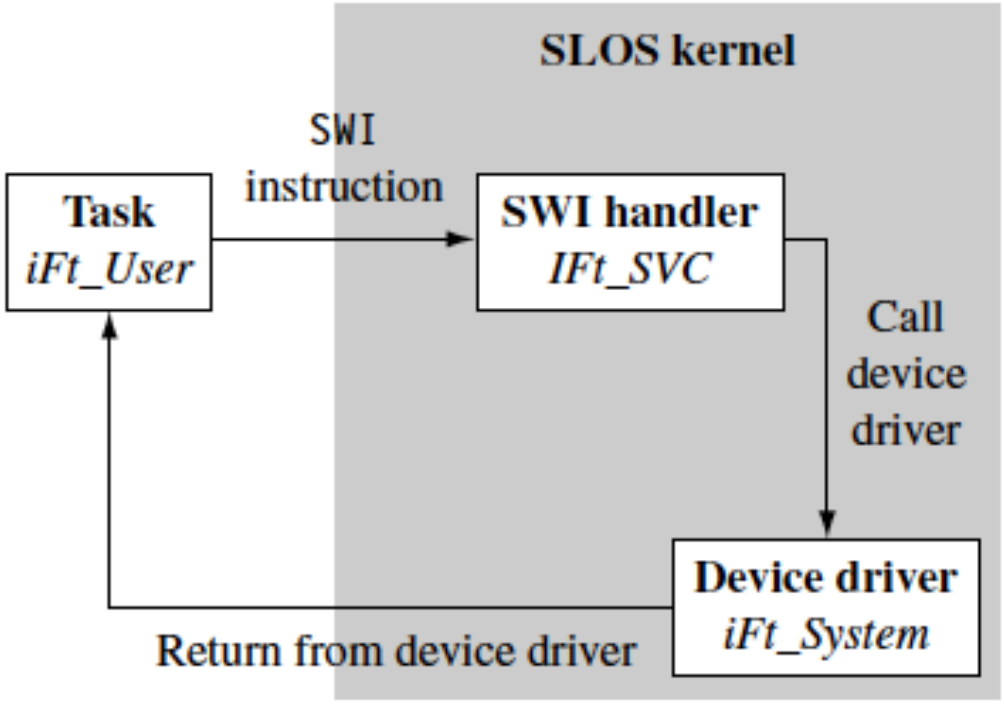
Step 5: Device Driver Framework

```
host = eventIODeviceOpen(&serial,DEVICE_SERIAL_E7T,COM1);

  if (host==0)
  {
    /* ...error device driver not found...*/
  }

  switch (serial)
  {
  case DEVICE_IN_USE:
  case DEVICE_UNKNOWN:
   /* ...problem with device... */
  }
```

Step 5: Device Driver Framework (Cont.)

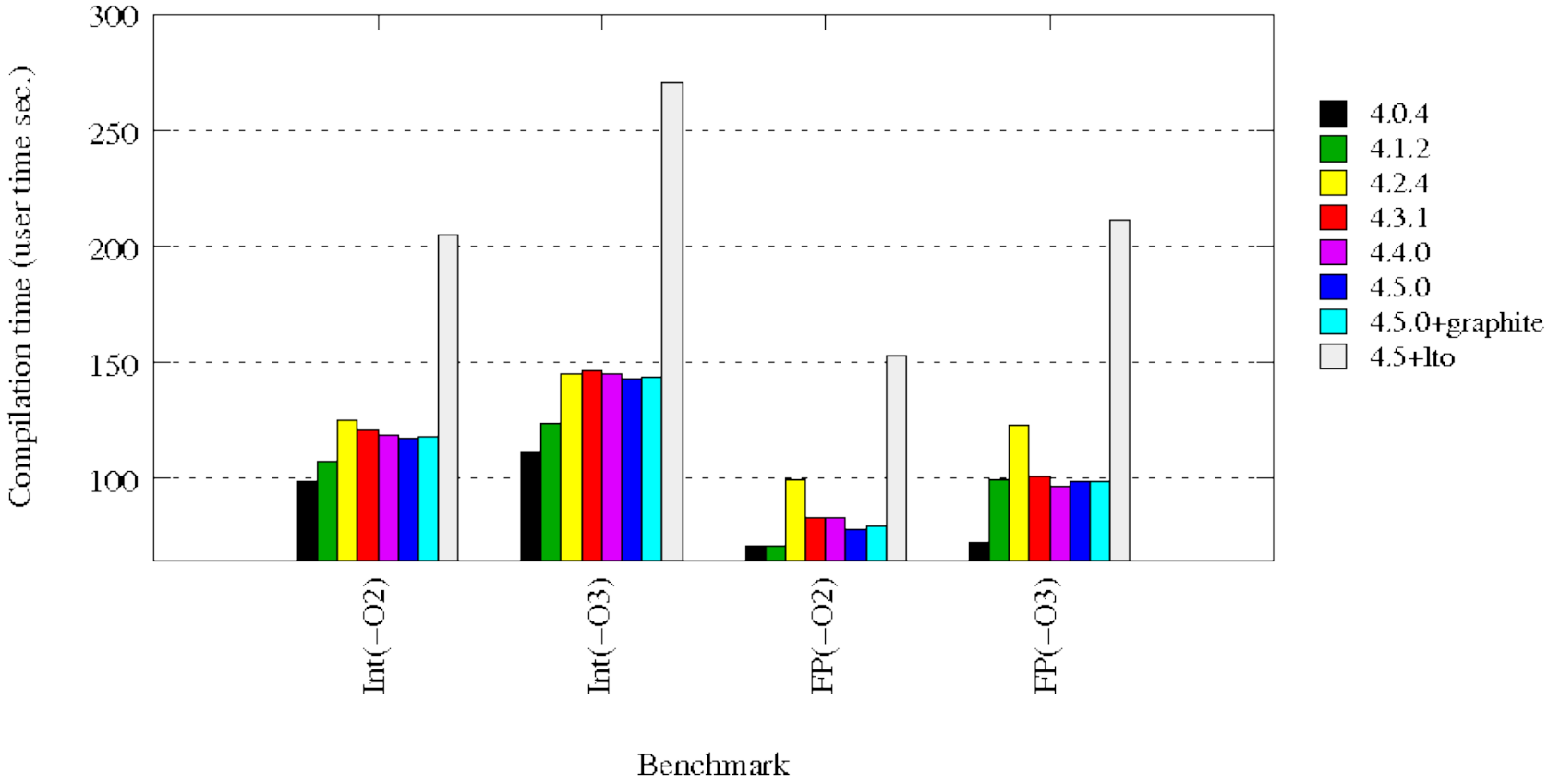
```
PRE     r0 = Event_IODeviceOpen (unsigned int)
        r1 = &serial (UID *u)
        r2 = DEVICE_SERIAL_E7T (unsigned int major)
        r3 = COM1 (unsigned int minor)

        SWI     5075

POST    r1 = The data pointed to by the UID pointer is updated
```

Step 5: Device Driver Framework (Cont.)

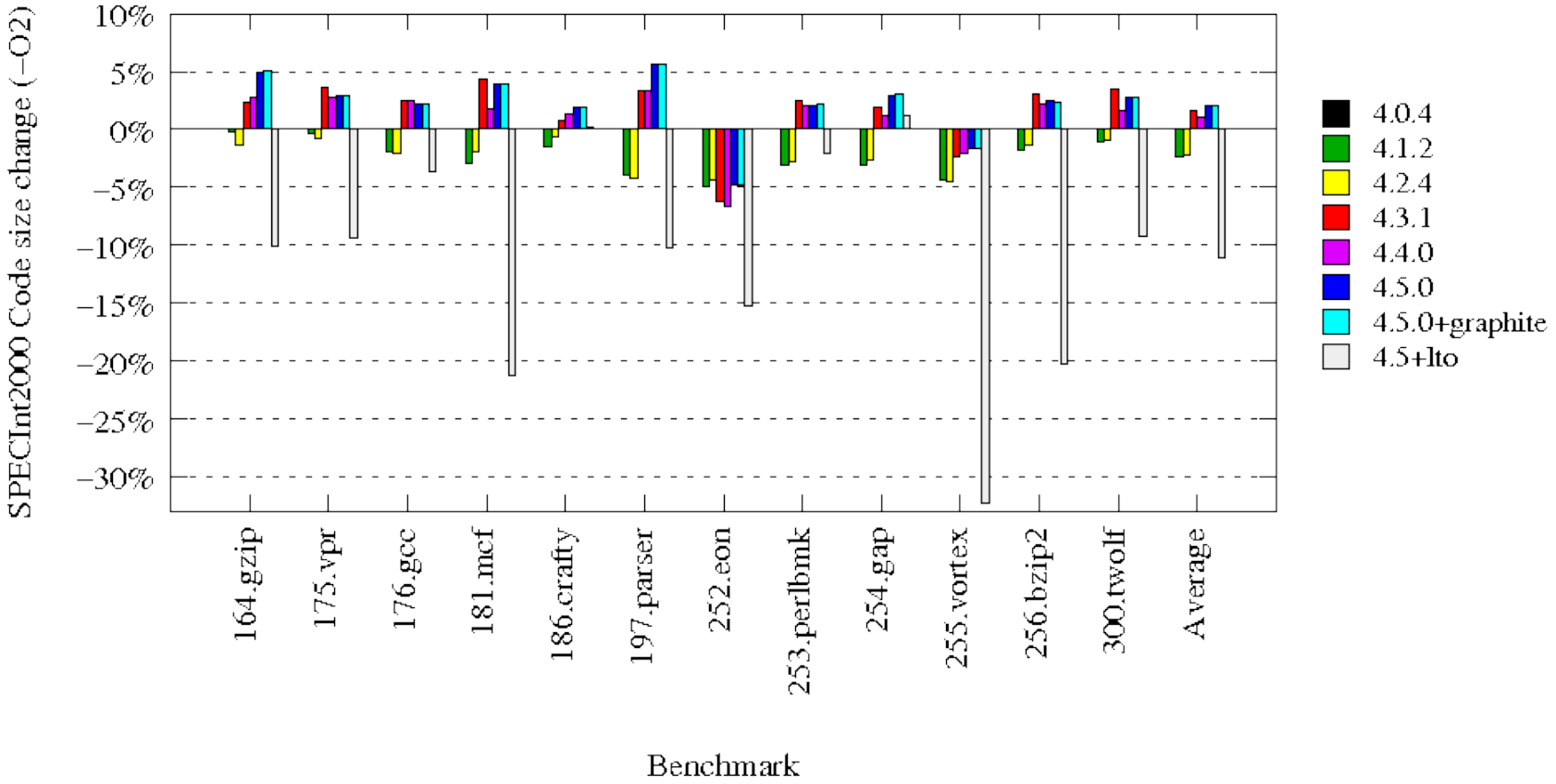


Efficient C Programming

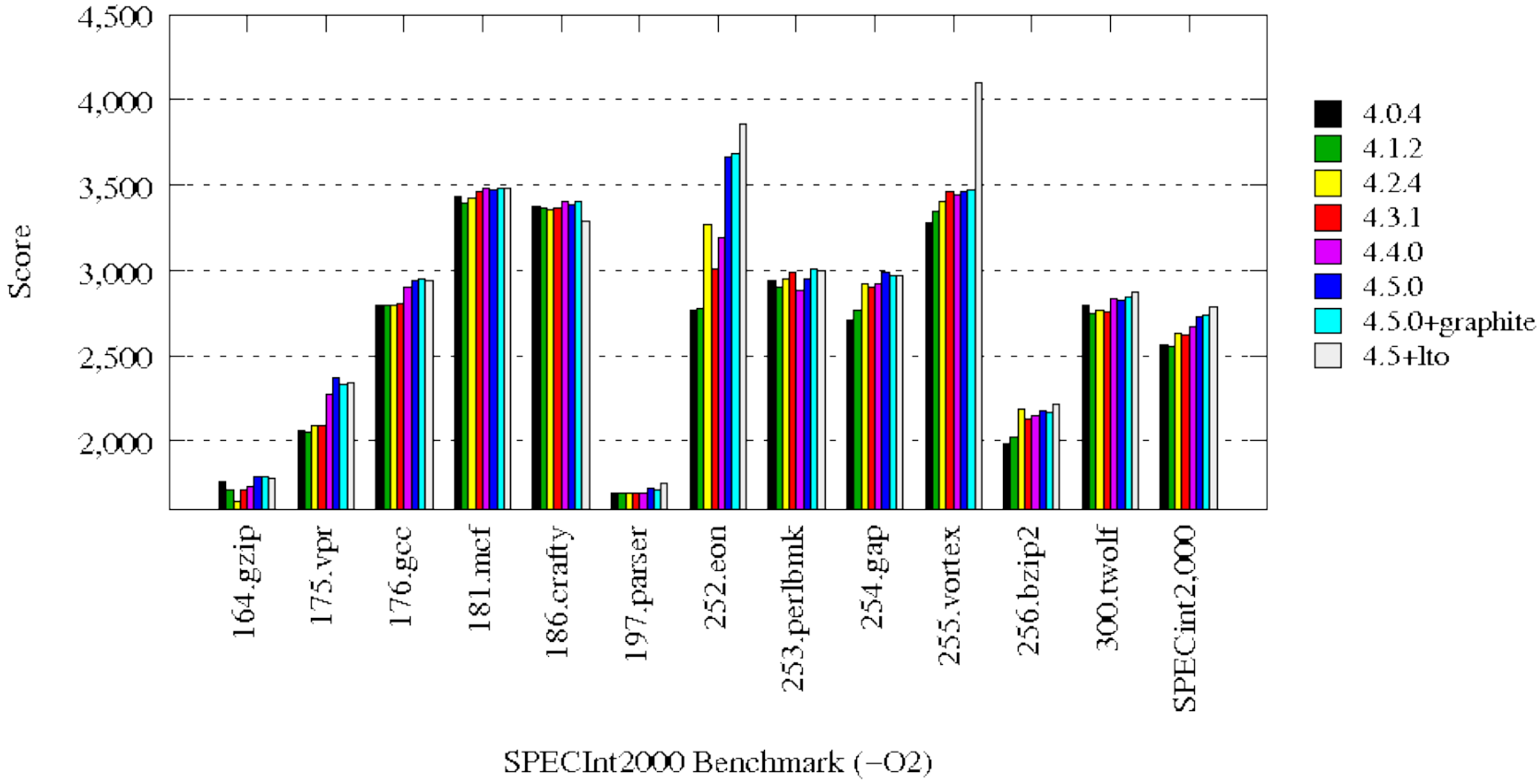
Compiler Versions



Compiler Versions (Cont.)



Compiler Versions (Cont.)



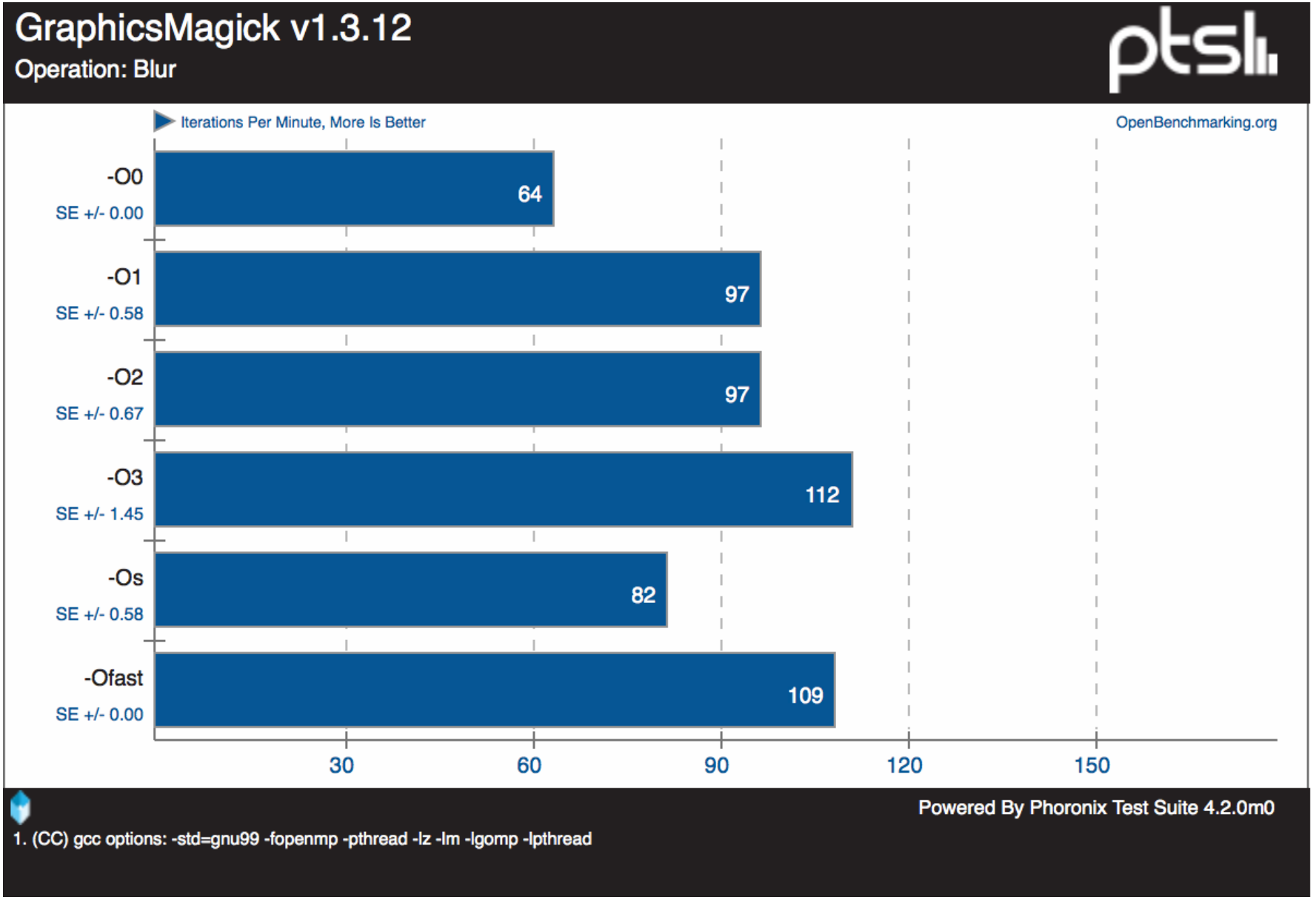
Compiler Options

- -O0
  - Reduce compilation time and make debugging produce the expected results. This is the default
- -O1
  - Optimizing compilation takes somewhat more time, and a lot more memory for a large function
- -O2
  - Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff
- -O3
  - turns on all optimizations specified by -O2 and also turns on the -finline-functions, -funswitch-loops, -fpredictive-commoning, …

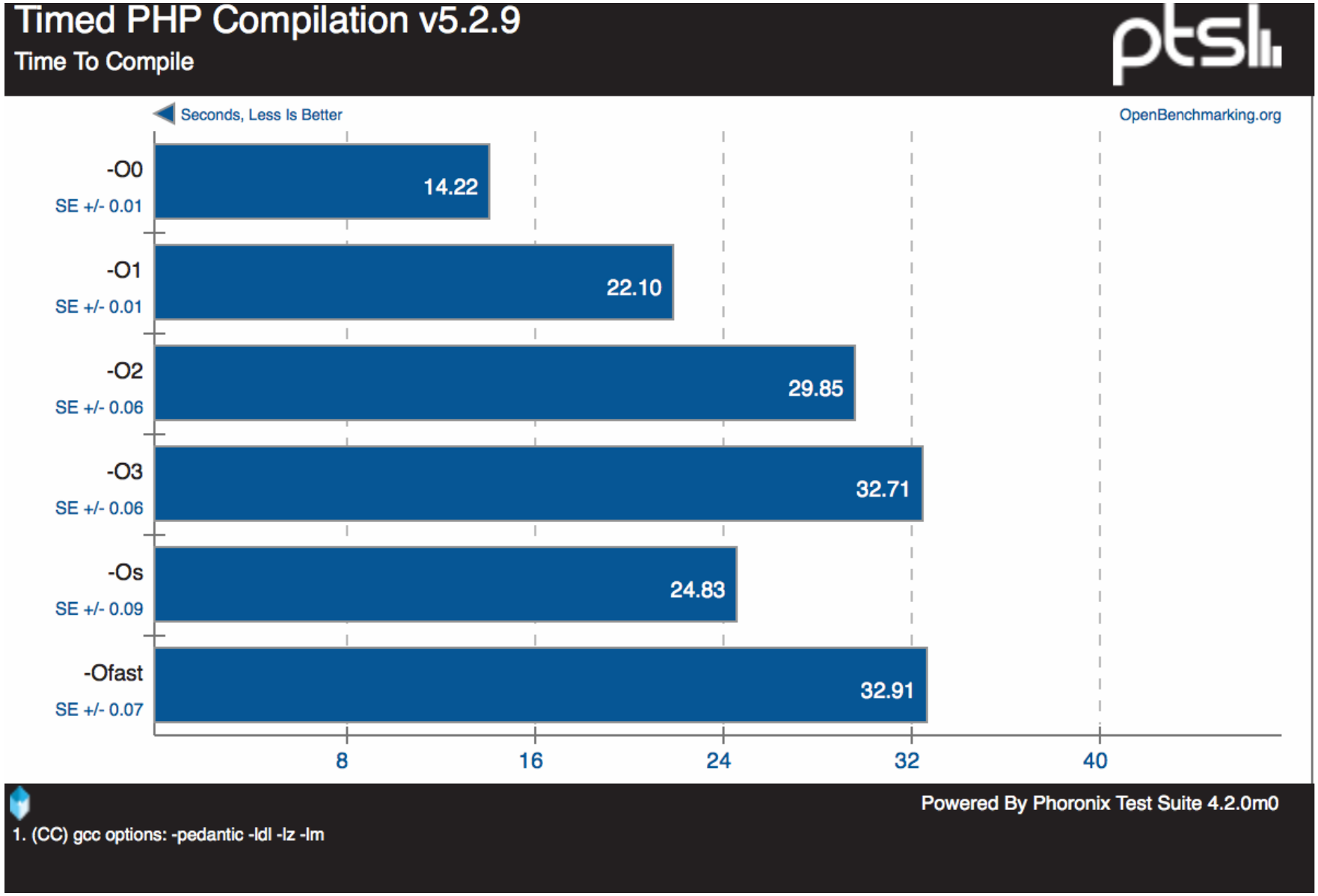
Compiler Options (Cont.)

- Os
  - Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size
- -Ofast
  - enables all -O3 optimizations. It also enables optimizations that are not valid for all standard-compliant programs

Performance



Compilation Time



Examples

- What are the problems?

```
void memclr(char *data, int N)
{
  for (; N>0; N--)
  {
    *data=0;
    data++;
  }
}
```

Technologies for Better C Code

- Basic C Data Types
- C Looping Structures
- Register Allocation
- Function Calls
- Pointer Aliasing
- Structure Arrangement
- Bit-fields
- Unaligned Data and Endianness
- Division
- Floating Point
- Inline Functions and Inline Assembly

Local Variable Types

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

  for (i=0; i<64; i++)
  {
    sum += data[i];
  }
  return sum;
}
```

```
checksum_v1
        MOV     r2,r0                   ; r2 = data
        MOV     r0,#0                   ; sum = 0
        MOV     r1,#0                   ; i = 0
checksum_v1_loop
        LDR     r3,[r2,r1,LSL #2]       ; r3 = data[i]
        ADD     r1,r1,#1                ; r1 = i+1
        AND     r1,r1,#0xff             ; i = (char)r1
        CMP     r1,#0x40                ; compare i, 64
        ADD     r0,r3,r0                ; sum += r3
        BCC     checksum_v1_loop        ; if (i<64) loop
        MOV     pc,r14                  ; return sum
```

Local Variable Types

```
checksum_v2
        MOV     r2,r0                   ; r2 = data
        MOV     r0,#0                   ; sum = 0
        MOV     r1,#0                   ; i = 0
checksum_v2_loop
        LDR     r3,[r2,r1,LSL #2]       ; r3 = data[i]
        ADD     r1,r1,#1                ; r1++
        CMP     r1,#0x40                ; compare i, 64
        ADD     r0,r3,r0                ; sum += r3
        BCC     checksum_v2_loop        ; if (i<64) goto loop
        MOV     pc,r14                  ; return sum
```

Local Variable Types

```
short checksum_v3(short *data)
{
  unsigned int i;
  short sum=0;

  for (i=0; i<64; i++)
  {
    sum = (short)(sum + data[i]);
  }
  return sum;
}
```

```
checksum_v3
        MOV     r2,r0               ; r2 = data
        MOV     r0,#0               ; sum = 0
        MOV     r1,#0               ; i = 0
checksum_v3_loop
        ADD     r3,r2,r1,LSL #1     ; r3 = &data[i]
        LDRH    r3,[r3,#0]          ; r3 = data[i]
        ADD     r1,r1,#1            ; i++
        CMP     r1,#0x40            ; compare i, 64
        ADD     r0,r3,r0            ; r0 = sum + r3
        MOV     r0,r0,LSL #16
        MOV     r0,r0,ASR #16       ; sum = (short)r0
        BCC     checksum_v3_loop    ; if (i<64) goto loop
        MOV     pc,r14              ; return sum
```

Local Variable Types

```
short checksum_v4(short *data)
{
  unsigned int i;
  int sum=0;

  for (i=0; i<64; i++)
  {
     sum += *(data++);
  }
  return (short)sum;
}
```

```
checksum_v4
        MOV     r2,#0               ; sum = 0
        MOV     r1,#0               ; i = 0
checksum_v4_loop
        LDRSH   r3,[r0],#2          ; r3 = *(data++)
        ADD     r1,r1,#1            ; i++
        CMP     r1,#0x40            ; compare i, 64
        ADD     r2,r3,r2            ; sum += r3
        BCC     checksum_v4_loop    ; if (sum<64) goto loop
        MOV     r0,r2,LSL #16
        MOV     r0,r0,ASR #16       ; r0 = (short)sum
        MOV     pc,r14              ; return r0
```

Functional Argument Types

```
add_v1
        ADD     r0,r0,r1,ASR #1     ; r0 = (int)a + ((int)b>>1)
        MOV     r0,r0,LSL #16
        MOV     r0,r0,ASR #16       ; r0 = (short)r0
        MOV     pc,r14              ; return r0
```

```
short add_v1(short a, short b)
{
  return a + (b>>1);
}
```

```
add_v1_gcc
        MOV     r0, r0, LSL #16
        MOV     r1, r1, LSL #16
        MOV     r1, r1, ASR #17         ; r1 = (int)b>>1
        ADD     r1, r1, r0, ASR #16     ; r1 += (int)a
        MOV     r1, r1, LSL #16
        MOV     r0, r1, ASR #16         ; r0 = (short)r1
        MOV     pc, lr                  ; return r0
```

The Efficient Use of C Types

- For local variables held in registers, don’t use a char or short type unless 8-bit or 16-bit modular arithmetic is necessary. Use the signed or unsigned int types instead. Unsigned types are faster when you use divisions
- For array entries and global variables held in main memory, use the type with the smallest size possible to hold the required data. This saves memory footprint

The Efficient Use of C Types (Cont.)

- Use explicit casts when reading array entries or global variables into local variables, or writing local variables out to array entries
- Avoid implicit or explicit narrowing casts in expressions because they usually cost extra cycles
- Avoid char and short types for function arguments or return values. Instead use the int type even if the range of the parameter is smaller. This prevents the compiler performing unnecessary casts

Loops with a Fixed Number of Iterations

```
int checksum_v5(int *data)
{
  unsigned int i;
  int sum=0;

  for (i=0; i<64; i++)
  {
    sum += *(data++);
  }
  return sum;
}
```

```
checksum_v5
        MOV     r2,r0               ; r2 = data
        MOV     r0,#0               ; sum = 0
        MOV     r1,#0               ; i = 0
checksum_v5_loop
        LDR     r3,[r2],#4          ; r3 = *(data++)
        ADD     r1,r1,#1            ; i++
        CMP     r1,#0x40            ; compare i, 64
        ADD     r0,r3,r0            ; sum += r3
        BCC     checksum_v5_loop    ; if (i<64) goto loop
        MOV     pc,r14              ; return sum
```

Loops with a Fixed Number of Iterations

```
int checksum_v6(int *data)
{
  unsigned int i;
  int sum=0;

  for (i=64; i!=0; i--)
  {
    sum += *(data++);
  }
  return sum;
}
```

```
checksum_v6
        MOV     r2,r0               ; r2 = data
        MOV     r0,#0               ; sum = 0
        MOV     r1,#0x40            ; i = 64
checksum_v6_loop
        LDR     r3,[r2],#4          ; r3 = *(data++)
        SUBS    r1,r1,#1            ; i-- and set flags
        ADD     r0,r3,r0            ; sum += r3
        BNE     checksum_v6_loop    ; if (i!=0) goto loop
        MOV     pc,r14              ; return sum
```

Loops Using A Variable Number of Iterations

```
int checksum_v7(int *data, unsigned int N)
{
  int sum=0;

  for (; N!=0; N--)
  {
    sum += *(data++);
  }
  return sum;
}
```

```
checksum_v7
        MOV     r2,#0               ; sum = 0
        CMP     r1,#0               ; compare N, 0
        BEQ     checksum_v7_end     ; if (N==0) goto end
checksum_v7_loop
        LDR     r3,[r0],#4          ; r3 = *(data++)
        SUBS    r1,r1,#1            ; N-- and set flags
        ADD     r2,r3,r2            ; sum += r3
        BNE     checksum_v7_loop    ; if (N!=0) goto loop
checksum_v7_end
        MOV     r0,r2               ; r0 = sum
        MOV     pc,r14              ; return r0
```

Loops Using A Variable Number of Iterations

```
int checksum_v8(int *data, unsigned int N)
{
  int sum=0;

  do
  {
    sum += *(data++);
  } while (--N!=0);
  return sum;
}
```

```
checksum_v8
        MOV     r2,#0               ; sum = 0
checksum_v8_loop
        LDR     r3,[r0],#4          ; r3 = *(data++)
        SUBS    r1,r1,#1            ; N-- and set flags
        ADD     r2,r3,r2            ; sum += r3
        BNE     checksum_v8_loop    ; if (N!=0) goto loop
        MOV     r0,r2               ; r0 = sum
        MOV     pc,r14              ; return r0
```

Loop Unrolling

```
int checksum_v9(int *data, unsigned int N)
{
  int sum=0;
  do
  {
    sum += *(data++);
    sum += *(data++);
    sum += *(data++);
    sum += *(data++);
    N -= 4;
  } while ( N!=0);
  return sum;
}
```

```
checksum_v9
        MOV r2,#0       ; sum = 0
checksum_v9_loop
        LDR     r3,[r0],#4      ; r3 = *(data++)
        SUBS    r1,r1,#4        ; N -= 4 & set flags
        ADD     r2,r3,r2        ; sum += r3
        LDR     r3,[r0],#4      ; r3 = *(data++)
        ADD     r2,r3,r2        ; sum += r3
        LDR     r3,[r0],#4      ; r3 = *(data++)
        ADD     r2,r3,r2        ; sum += r3
        LDR     r3,[r0],#4      ; r3 = *(data++)
        ADD     r2,r3,r2        ; sum += r3
        BNE     checksum_v9_loop ; if (N!=0) goto loop
        MOV     r0,r2           ; r0 = sum
        MOV     pc,r14          ; return r0
```

Writing Loops Efficiently

- Use loops that count down to zero. Then the compiler does not need to allocate a register to hold the termination value, and the comparison with zero is free
- Use unsigned loop counters by default and the continuation condition i!=0 rather than i>0. This will ensure that the loop overhead is only two instructions
- Use do-while loops rather than for loops when you know the loop will iterate at least once. This saves the compiler checking to see if the loop count is zero

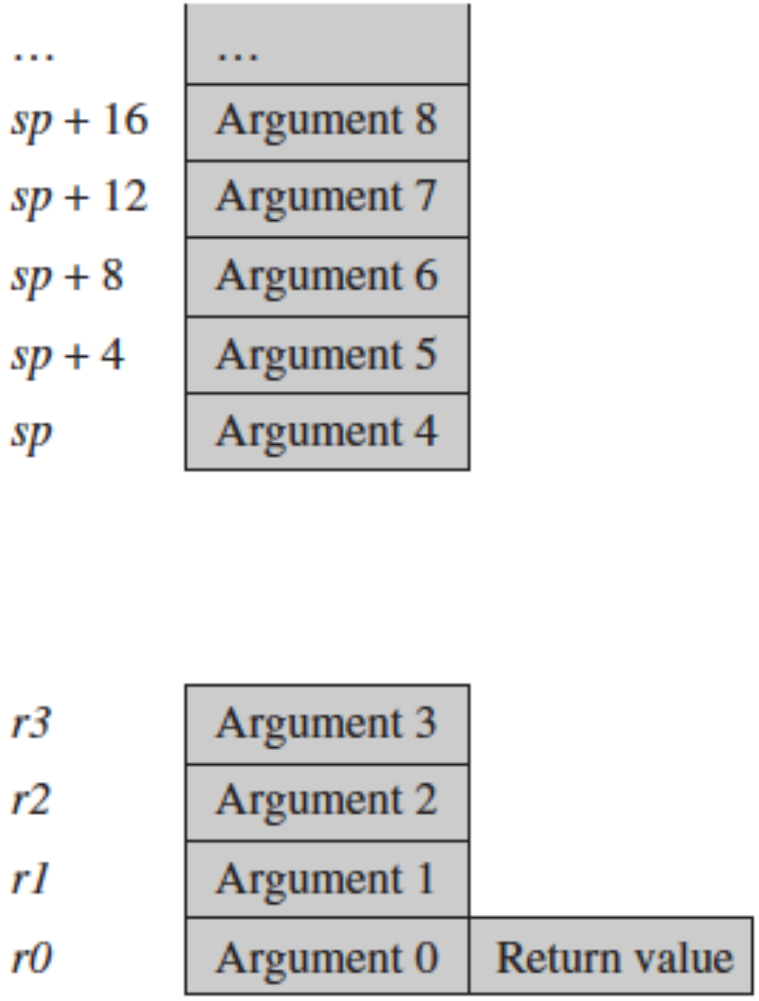
Writing Loops Efficiently (Cont.)

- Unroll important loops to reduce the loop overhead. Do not overunroll. If the loop overhead is small as a proportion of the total, then unrolling will increase code size and hurt the performance of the cache
- Try to arrange that the number of elements in arrays are multiples of four or eight. You can then unroll loops easily by two, four, or eight times without worrying about the leftover array elements

Register Allocations

- Try to limit the number of local variables in the internal loop of functions to 12. The compiler should be able to allocate these to ARM registers
- You can guide the compiler as to which variables are important by ensuring these variables are used within the innermost loop

Function Calls



```
char *queue_bytes_v1(
  char *Q_start,          /* Queue buffer start address */
  char *Q_end,            /* Queue buffer end address */
  char *Q_ptr,            /* Current queue pointer position */
  char *data,             /* Data to insert into the queue */
  unsigned int N)         /* Number of bytes to insert */
{
  do
  {
    *(Q_ptr++) = *(data++);

    if (Q_ptr == Q_end)
    {
      Q_ptr = Q_start;
    }
  } while (--N);
  return Q_ptr;
}
```

```
queue_bytes_v1
        STR     r14,[r13,#-4]!   ; save lr on the stack
        LDR     r12,[r13,#4]     ; r12 = N
queue_v1_loop
        LDRB    r14,[r3],#1      ; r14 = *(data++)
        STRB    r14,[r2],#1      ; *(Q_ptr++) = r14
        CMP     r2,r1            ; if (Q_ptr == Q_end)
        MOVEQ   r2,r0            ;    {Q_ptr = Q_start;}
        SUBS    r12,r12,#1       ; --N and set flags
        BNE     queue_v1_loop    ; if (N!=0) goto loop
        MOV     r0,r2            ; r0 = Q_ptr
        LDR     pc,[r13],#4      ; return r0
```

```
typedef struct {
  char *Q_start;          /* Queue buffer start address */
  char *Q_end;            /* Queue buffer end address */
  char *Q_ptr;            /* Current queue pointer position */
} Queue;

void queue_bytes_v2(Queue *queue, char *data, unsigned int N)
{
  char *Q_ptr = queue->Q_ptr;
  char *Q_end = queue->Q_end;
                          queue_bytes_v2
  do                          STR     r14,[r13,#-4]!   ; save lr on the stack
  {                           LDR     r3,[r0,#8]       ; r3 = queue->Q_ptr
    *(Q_ptr++) = *(data++);   LDR     r14,[r0,#4]      ; r14 = queue->Q_end
                          queue_v2_loop
    if (Q_ptr == Q_end)       LDRB    r12,[r1],#1      ; r12 = *(data++)
    {                         STRB    r12,[r3],#1      ; *(Q_ptr++) = r12
      Q_ptr = queue->Q_start; CMP     r3,r14           ; if (Q_ptr == Q_end)
    }                         LDREQ   r3,[r0,#0]       ;    Q_ptr = queue->Q_start
  } while (--N);              SUBS    r2,r2,#1         ; --N and set flags
  queue->Q_ptr = Q_ptr;       BNE     queue_v2_loop    ; if (N!=0) goto loop
}                             STR     r3,[r0,#8]       ; queue->Q_ptr = r3
                              LDR     pc,[r13],#4      ; return
```

Calling Functions Efficiently

- Try to restrict functions to four arguments. This will make them more efficient to call. Use structures to group related arguments and pass structure pointers instead of multiple arguments
- Define small functions in the same source file and before the functions that call them. The compiler can then optimize the function call or inline the small function
- Critical functions can be inlined using the __inline keyword

Pointer Aliasing

```
void timers_v1(int *timer1, int *timer2, int *step)
{
  *timer1 += *step;
  *timer2 += *step;
}
```

```
timers_v1
        LDR     r3,[r0,#0]          ; r3 = *timer1
        LDR     r12,[r2,#0]         ; r12 = *step
        ADD     r3,r3,r12           ; r3 += r12
        STR     r3,[r0,#0]          ; *timer1 = r3
        LDR     r0,[r1,#0]          ; r0 = *timer2
        LDR     r2,[r2,#0]          ; r2 = *step
        ADD     r0,r0,r2            ; r0 += r2
        STR     r0,[r1,#0]          ; *timer2 = t0
        MOV     pc,r14              ; return
```

Pointer Aliasing (Cont.)

```
typedef struct {int step;} State;
typedef struct {int timer1, timer2;} Timers;

void timers_v2(State *state, Timers *timers)
{
  timers->timer1 += state->step;
  timers->timer2 += state->step;
}
```

```
void timers_v3(State *state, Timers *timers)
{
      int step = state->step;

      timers->timer1 += step;
      timers->timer2 += step;
}
```

Avoiding Pointer Aliasing

- Do not rely on the compiler to eliminate common subexpressions involving memory accesses. Instead create new local variables to hold the expression. This ensures the expression is evaluated only once
- Avoid taking the address of local variables. The variable may be inefficient to access from then on

Structure Arrangement

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

Address	+3	+2	+1	+0
+0	pad	pad	pad	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]
+8	d[15,8]	d[7,0]	pad	c

Structure Arrangement (Cont.)

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

Address	+3	+2	+1	+0
+0	d[15,8]	d[7,0]	c	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]

Structure Arrangement (Cont.)

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

Address	+3	+2	+1	+0
+0	b[23,16]	b[15,8]	b[7,0]	a
+4	d[15,8]	d[7,0]	c	b[31,24]

Efficient Structure Arraignment

- Lay structures out in order of increasing element size. Start the structure with the smallest elements and finish with the largest
- Avoid very large structures. Instead use a hierarchy of smaller structures
- For portability, manually add padding (that would appear implicitly) into API structures so that the layout of the structure does not depend on the compiler
- Beware of using enum types in API structures. The size of an enum type is compiler dependent

Bit Fields

```
void dostageA(void);
void dostageB(void);
void dostageC(void);

typedef struct {
  unsigned int stageA : 1;
  unsigned int stageB : 1;
  unsigned int stageC : 1;
} Stages_v1;

void dostages_v1(Stages_v1 *stages)
{
  if (stages->stageA)
  {
    dostageA();
  }
  if (stages->stageB)
  {
    dostageB();
  }
  if (stages->stageC)
  {
    dostageC();
  }
}
```

Bit Fields (Cont.)

```
dostages_v1
        STMFD    r13!,{r4,r14}     ; stack r4, lr
        MOV      r4,r0             ; move stages to r4
        LDR      r0,[r0,#0]        ; r0 = stages bitfield
        TST      r0,#1             ; if (stages->stageA)
        BLNE     dostageA          ;    {dostageA();}
        LDR      r0,[r4,#0]        ; r0 = stages bitfield
        MOV      r0,r0,LSL #30     ; shift bit 1 to bit 31
        CMP      r0,#0             ; if (bit31)
        BLLT     dostageB          ;    {dostageB();}
        LDR      r0,[r4,#0]        ; r0 = stages bitfield
        MOV      r0,r0,LSL #29     ; shift bit 2 to bit 31
        CMP      r0,#0             ; if (!bit31)
        LDMLTFD  r13!,{r4,r14}     ;    return
        BLT      dostageC          ; dostageC();
        LDMFD    r13!,{r4,pc}      ; return
```

Bit Fields (Cont.)

```
typedef unsigned long Stages_v2;

#define STAGEA (1ul<<0)
#define STAGEB (1ul<<1)
#define STAGEC (1ul<<2)

void dostages_v2(Stages_v2 *stages_v2)
{
  Stages_v2 stages = *stages_v2;

  if (stages & STAGEA)
  {
    dostageA();
  }
  if (stages & STAGEB)
  {
    dostageB();
  }
  if (stages & STAGEC)
  {
    dostageC();
  }
}
```

Bit Fields (Cont.)

```
dostages_v2
        STMFD    r13!,{r4,r14}   ; stack r4, lr
        LDR      r4,[r0,#0]      ; stages = *stages_v2
        TST      r4,#1           ; if (stage & STAGEA)
        BLNE     dostageA        ;    {dostageA();}
        TST      r4,#2           ; if (stage & STAGEB)
        BLNE     dostageB        ;    {dostageB();}
        TST      r4,#4           ; if (!(stage & STAGEC))
        LDMNEFD  r13!,{r4,r14}   ; return;
        BNE      dostageC        ; dostageC();
        LDMFD    r13!,{r4,pc}    ; return
```

Bit-fields

- Avoid using bit-fields. Instead use #define or enum to define mask values
- Test, toggle, and set bit-fields using integer logical AND, OR, and exclusive OR operations with the mask values. These operations compile efficiently, and you can test, toggle, or set multiple fields at the same time

Unaligned Data and Endiannes

Little-endian configuration.

Instruction	Width (bits)	b31..b24	b23..b16	b15..b8	b7..b0
LDRB	8	0	0	0	B(A)
LDRSB	8	S(A)	S(A)	S(A)	B(A)
STRB	8	X	X	X	B(A)
LDRH	16	0	0	B(A+1)	B(A)
LDRSH	16	S(A+1)	S(A+1)	B(A+1)	B(A)
STRH	16	X	X	B(A+1)	B(A)
LDR/STR	32	B(A+3)	B(A+2)	B(A+1)	B(A)

Unaligned Data and Endiannes (Cont.)

Big-endian configuration.

Instruction	Width (bits)	b31..b24	b23..b16	b15..b8	b7..b0
LDRB	8	0	0	0	B(A)
LDRSB	8	S(A)	S(A)	S(A)	B(A)
STRB	8	X	X	X	B(A)
LDRH	16	0	0	B(A)	B(A+1)
LDRSH	16	S(A)	S(A)	B(A)	B(A+1)
STRH	16	X	X	B(A)	B(A+1)
LDR/STR	32	B(A)	B(A+1)	B(A+2)	B(A+3)

Endianness and Alignment

- Avoid using unaligned data if you can
- Use the type char * for data that can be at any byte alignment. Access the data by reading bytes and combining with logical operations. Then the code won't depend on alignment or ARM endianness configuration
- For fast access to unaligned structures, write different variants according to pointer alignment and processor endianness

Division

```
offset = (offset + increment) % buffer_size;
```

```
offset += increment;
if (offset>=buffer_size)
{
   offset -= buffer_size;
}
```

Converting Divides into Multiples

```
void scale(
  unsigned int *dest,              /* destination for the scale data */
  unsigned int *src,               /* source unscaled data */
  unsigned int d,                  /* denominator to divide by */
  unsigned int N)                  /* data length */
{
  unsigned int s = 0xFFFFFFFFu / d;
```

Converting Divides into Multiples (Cont.)

```
do
{
  unsigned int n, q, r;

  n = *(src++);
  q = (unsigned int)(((unsigned long long)n * s) >> 32);
  r = n-q * d;
  if (r >= d)
  {
     q++;
  }
  *(dest++) = q;
} while (--N);
}
```

Unsigned Division by a Constant

```
unsigned int udiv_by_const(unsigned int n, unsigned int d)
{
  unsigned int s,k,q;

  /* We assume d!=0 */

  /* first find k such that (1<<k) <= d < (1<<(k+1)) */
  for (k=0; d/2>=(1u<<k); k++);

  if (d==1u<<k)
  {
    /* we can implement the divide with a shift */
    return n>>k;
  }

  /* d is in the range (1<<k) < d < (1<<(k+1)) */
  s = (unsigned int)(((1ull<<(32+k))+(1ull<<k))/d);
```

Unsigned Division by a Constant (Cont.)

```
if ((unsigned long long)s*d >= (1ull<<(32+k)))
{
     /* n/d = (n*s)>>(32+k) */
     q = (unsigned int)(((unsigned long long)n*s)>>32);
     return q>>k;
}

/* n/d = (n*s+s)>>(32+k) */


 q = (unsigned int)(((unsigned long long)n*s + s)>>32);
 return q>>k;
}
```

Division

- Avoid divisions as much as possible. Do not use them for circular buffer handling
- If you can't avoid a division, then try to take advantage of the fact that divide routines often generate the quotient n/d and modulus n%d together
- To repeatedly divide by the same denominator d , calculate s = $(2^k - 1)/d$ in advance. You can replace the divide of a k -bit unsigned integer by d with a 2k -bit multiply by s

Division (Cont.)

- To divide unsigned n < 2^N by an unsigned constant d , you can find a 32-bit unsigned s and shift k such that n/d is either (ns) >> (N + k) or (ns + s) >> (N + k). The choice depends only on d . There is a similar result for signed divisions

Floating Point

- The majority of ARM processor implementations do not provide hardware floating-point support, which saves on power and area
- In practice, this means that the C compiler converts every floating-point operation into a subroutine call. The C library contains subroutines to simulate floating-point behavior using integer arithmetic. This code is written in highly optimized assembly
- If you need fast execution and fractional values, you should use fixed-point or blockfloating algorithms

Inline Functions and Inline Assembly

- Use inline functions to declare new operations or primitives not supported by the C compiler
- Use inline assembly to access ARM instructions not supported by the C compiler. Examples are coprocessor instructions or ARMv5E extensions.

Efficient Assembly Programming

Technologies to Write Efficient Assembly Code

- Instruction Scheduling
- Register Allocation
- Conditional Execution
- Bit Manipulation
- Efficient Switches
- Handling Unaligned Data

Write Assembly Code

```
#include <stdio.h>

int square(int i);

int main(void)
{
  int i;

  for (i=0; i<10; i++)
  {
    printf("Square of %d is %d\n", i, square(i));
  }
}

int square(int i)
{
  return i*i;
}
```

Write Assembly Code (Cont.)

```
        AREA    |.text|, CODE, READONLY

        EXPORT  square

        ; int square(int i)
square
        MUL   r1, r0, r0   ; r1 = r0 * r0
        MOV   r0, r1       ; r0 = r1
        MOV   pc, lr       ; return r0
        END
```

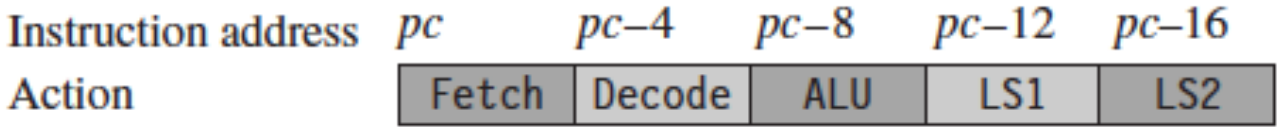
Write Assembly Code (Cont.)

```
        AREA    |.text|, CODE, READONLY

        EXPORT  main

        IMPORT  |Lib$$Request$$armlib|, WEAK
        IMPORT  __main    ; C library entry
        IMPORT  printf    ; prints to stdout
i       RN 4

        ; int main(void)

main
        STMFD   sp!, {i, lr}
        MOV     i, #0
```

Write Assembly Code (Cont.)

```
loop
        ADR     r0, print_string
        MOV     r1, i
        MUL     r2, i, i
        BL      printf
        ADD     i, i, #1
        CMP     i, #10
        BLT     loop
        LDMFD   sp!, {i, pc}

print_string
        DCB      "Square of %d is %d\n", 0

        END
```

Instruction Scheduling




```
ADD   r0, r0, r1
ADD   r0, r0, r2
```

```
LDR   r1, [r2, #4]
ADD   r0, r0, r1
```

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	...	ADD	LDR	...	
Cycle 2		...	*ADD*	LDR	...
Cycle 3		...	ADD	—	LDR

Instruction Scheduling (Cont.)

```
        LDRB    r1, [r2, #1]
        ADD     r0, r0, r2
        EOR     r0, r0, r1
```

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	EOR	ADD	LDRB	...	
Cycle 2	...	EOR	ADD	LDRB	...
Cycle 3		...	*EOR*	ADD	LDRB
Cycle 4		...	EOR	—	ADD

```
        MOV     r1, #1
        B       case1
        AND     r0, r0, r1
        EOR     r2, r2, r3
        ...
case1
        SUB     r0, r0, r1
```

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	AND	B	MOV	...	
Cycle 2	EOR	AND	B	MOV	...
Cycle 3	SUB	—	—	B	MOV
Cycle 4	...	SUB	—	—	B
Cycle 5		...	SUB	—	—

Instruction Scheduling (Cont.)

- ARM cores have a pipeline architecture. The pipeline may delay the results of certain instructions for several cycles. If you use these results as source operands in a following instruction, the processor will insert stall cycles until the value is ready
- Load and multiply instructions have delayed results in many implementations
- You have two software methods available to remove interlocks following load instructions: You can preload so that loop i loads the data for loop i + 1, or you can unroll the loop and interleave the code for loops i and i + 1.

Register Allocation

- ARM has 14 available registers for general-purpose use: r0 to r12 and r14. The stack pointer r13 and program counter r15 cannot be used for general-purpose data. Operating system interrupts often assume that the user mode r13 points to a valid stack, so don’t be tempted to reuse r13.
- If you need more than 14 local variables, swap the variables out to the stack, working outwards from the innermost loop

Register Allocation (Cont.)

- Use register names rather than physical register numbers when writing assembly routines. This makes it easier to reallocate registers and to maintain the code
- To ease register pressure you can sometimes store multiple values in the same register. For example, you can store a loop counter and a shift in one register. You can also store multiple pixels in one register

Conditional Execution

```
if (i<10)
{
  c = i + '0';
}
else
{
  c = i + 'A'-10;
}
```

```
CMP     i, #10
ADDLO   c, i, #'0'
ADDHS   c, i, #'A'-10
```

Conditional Execution

- You can implement most if statements with conditional execution. This is more efficient than using a conditional branch

Loop Constructs

```
      MOV i, N
loop
      ; loop body goes here and i=N,N-1,...,1
      SUBS i, i, #1
      BGT  loop
```

```
      SUBS i, N, #1
loop
      ; loop body goes here and i=N-1,N-2,...,0
      SUBS i, i, #1
      BGE  loop
```

Loop Constructs (Cont.)

```
      MOV i, N
loop
      ; loop body goes here and iterates (round up)(N/3) times
      SUBS i, i, #3
      BGT  loop
```

Loop Constructs (Cont.)

- ARM requires two instructions to implement a counted loop: a subtract that sets flags and a conditional branch
- Unroll loops to improve loop performance. Do not overunroll because this will hurt cache performance. Unrolled loops may be inefficient for a small number of iterations. You can test for this case and only call the unrolled loop if the number of iterations is large

Loop Constructs (Cont.)

- Nested loops only require a single loop counter register, which can improve efficiency by freeing up registers for other uses
- ARM can implement negative and logarithmic indexed loops efficiently

Bit Manipulation

- The ARM can pack and unpack bits efficiently using logical operations and the barrel shifter
- To access bitstreams efficiently use a 32-bit register as a bitbuffer. Use a second register to keep track of the number of valid bits in the bitbuffer
- To decode bitstreams efficiently, use a lookup table to scan the next N bits of the bitstream. The lookup table can return codes of length at most N bits directly, or return an escape character for longer codes.

Efficient Switches

- Make sure the switch value is in the range $0 \leq x < N$ for some small N . To do this you may have to use a hashing function
- Use the switch value to index a table of function pointers or to branch to short sections of code at regular intervals. The second technique is position independent; the first isn’t

Handling Unaligned Data

- If performance is not an issue, access unaligned data using multiple byte loads and stores. This approach accesses data of a given endianness regardless of the pointer alignment and the configured endianness of the memory system
- If performance is an issue, then use multiple routines, with a different routine optimized for each possible array alignment. You can use the assembler MACRO directive to generate these routines automatically

Thank you