

Week 4. ARM Assembly Programming (Finale)

18-342: Fundamentals of Embedded Systems

Rajeev Gandhi

INI & ECE
Carnegie Mellon University

Carnegie Mellon

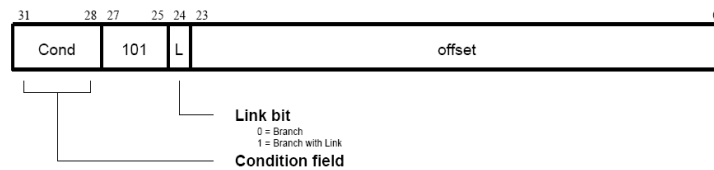


Overview of this Week

- Digging deeper
 - Limitations of branch instructions
 - Loading arbitrary 32-bit constants in registers
- Pipelining
 - Basics of pipelining
 - Pipelining on the ARM architecture
 - Effect of pipelining on the program counter
 - Advanced issues in pipelining
- ATPCS conventions
- Compiling and linking
- Dynamically/Statically linked libraries
- Executable and Linking Format (ELF)
- **Announcement:**
 - Quiz statistics: max/avg/min were 94.5/76.58/36
 - Standard deviation: 12.95

Digging Deeper

◆ Branch instructions



- When executing the instruction, the processor
 - Shifts the offset left two bits, sign extends it to 32 bits, and adds it to the `pc`
- This gives a 26-bit offset from the current instruction
- What is the maximum distance (in bytes) that we can jump to, in a branch?

3

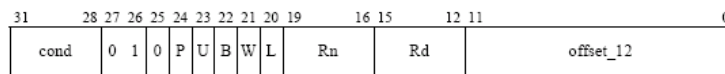
LDR Instruction

◆ `LDR rd, [rn, #offset]` $rd \leftarrow \text{mem}[rn + \text{offset}]$

- Decimal numbers prefixed by #

◆ `rd, rn` can be any register (`r0 – r15`)

◆ Binary encoding of LDR/STR instruction (with immediate offset addressing mode)



- distinguish between load (1) and store (0)
- add or subtract offset

4

LDR (contd.)

- ◆ LDR can be used for branches beyond the range of BL instruction
 - Example: LDR pc, [pc, #offset] pc ← mem[pc+offset]
 - The address of the branch should be stored in memory location **curraddr**
+8+offset

- ◆ Example

```
0x10000000    add r0, r1, r2
0x10000004    ldr pc, [pc, #4];
0x10000008    sub r1, r2, r3
0x1000000c    cmp r0, r1
0x10000010    0x20000000
...
...
Branch_target
0x20000000    str r5, [r13, #-4]!
...
...
```

- ◆ STR/LDR instructions can be used to save/restore registers on function entry/exit

5

Digging Deeper: Loading Constants

- ◆ There is no single instruction which will load a 32 bit immediate constant into a register without performing a data load from memory.
 - All ARM instructions are 32 bits long
 - ARM instructions do not use the instruction stream as data
- ◆ The data processing instruction format has 12 bits available for operand2
 - If used directly, this would only give a range of 4096 bytes
- ◆ Instead it is used to store 8-bit constants, giving a range of 0 - 255
- ◆ These 8 bits can then be rotated right through an even number of positions
 - This gives a much larger range of constants that can be directly loaded, though some constants will still need to be loaded from memory

6

Specific Example: MOV

- ◆ MOV or MVN data processing instruction can be used to load 8-bit numbers into registers
 - `MOV r0, #0x07 ; r0=0x00000007`
- ◆ Use MOV with the barrel shifter to load more than 8-bit numbers into registers
 - `MOV r0, #0x0F, #2 ; r0=0xC0000003`
- ◆ Remember operand2 in MOV instruction takes 12 bits
 - 8 bits are used for the immediate value
 - 4 bit rotate value (which should be an even number between 0 and 30)
- ◆ Rule to remember is “8-bits rotated right by an even number of bit positions”.

7

Loading Constants: Pseudo-Instruction LDR

- ◆ To allow larger constants to be loaded, the assembler offers a **pseudo-instruction**:
 - `LDR Rd, =const`
- ◆ This will either:
 - Produce a MOV or MVN instruction to generate the value (if possible)**or**
 - Generate a LDR instruction with a pc-relative address to read the constant from a *literal pool* (constant data area embedded in the code)
 - What is a literal pool?
 - Portion of memory used by the assembly code to store constants
- ◆ This is the recommended way of loading constants into a register

8

Loading Constants – Example

Original assembly code

```
LDR r0, =0x55555555
LDR r1, =0x4867
ADD r2, r1, r0
SWI 0x123456
```

Machine code generated by assembler

```
0x00000000: e59f0008
0x00000004: e59f1008
0x00000008: e0812000
0x0000000c: ef123456
0x00000010: 55555555
0x00000014: 0004867
```

- Assembler places the constant 0x55555555 at location 0x00000010
- and converts the pseudo-instruction to LDR r0, [pc, #8]
- Assembler places the constant 0x4867 at location 0x00000014
- and converts the pseudo-instruction to LDR r0, [pc, #8]

9

DETOUR

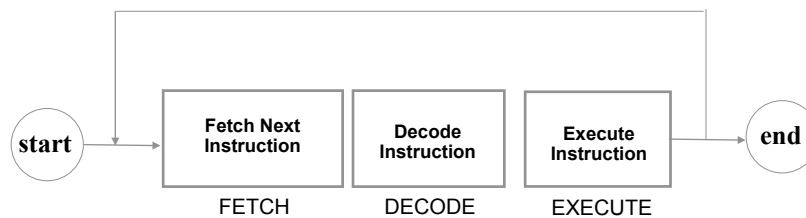


Pipelining

- Typical steps involved in executing an instruction
 - Fetch the instruction from memory
 - Decode the opcode of the instruction
 - Depending upon the kind of instruction
 - Read one or more operand registers from the register file
 - Execute the instruction
 - Perform ALU operation for data processing instruction
 - Compute effective address (base+offset) for a LDR/STR instruction
 - Compute the target address of a branch instruction
 - Access memory for LDR/STR instruction
 - Write the results of data processing instruction, LDR instruction in the destination register

Instruction Cycle on an ARMv6 Processor

- Three steps:
 - Fetch instruction
 - Decode instruction
 - Execute instruction (read registers, perform ALU operation, read/write memory, write to destination register)



Unpipelined Implementation of the instruction cycle

- Two possible implementations
 - One long clock cycle: fetch, decode and execute occur within the long clock cycle
 - A short(er) clock cycle where fetch occurs in one clock cycle, decode occurs in the next clock cycle and execute occurs in the next clock cycle
 - Processor can run at a higher (possibly 3 times higher) clock frequency but each instruction takes three clock cycle

Enter Pipelining

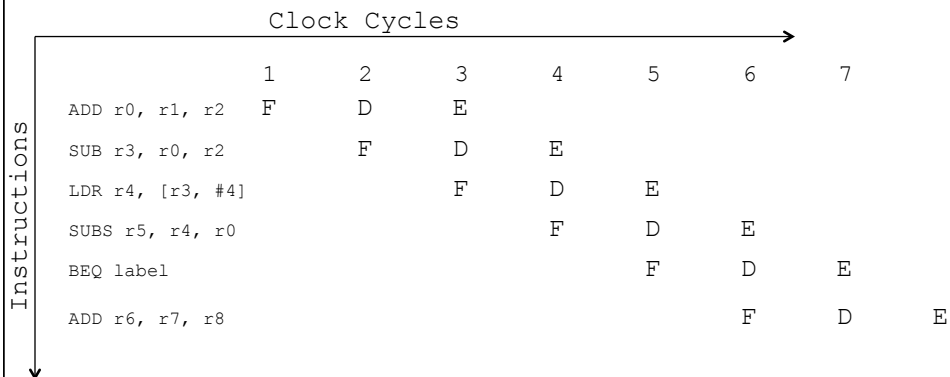
- Key observation in the second implementation (previous slide) of the processor
 - In Fetch cycle the decode and execute hardware is idle
 - In Decode cycle the fetch and execute hardware is idle
 - In Execute cycle the fetch and decode hardware is idle
- Pipelining
 - Fetch an instruction (*instr1*) in first clock cycle
 - While the decode hardware is decoding *instr1* (in 2nd clock cycle), fetch the next instruction *instr2*
 - While the execute hardware is executing *instr1* (in 3rd clock cycle), decode *instr2* and fetch another instruction *instr3*
- Several instructions can be processing simultaneously on a pipelined processor
- All hardware units are used in every clock cycle

Enter Pipelining (2)

```
ADD r0, r1, r2
SUB r3, r0, r2
LDR r4, [r3, #4]
SUBS r5, r4, r0
BEQ label
ADD r6, r7, r8
```

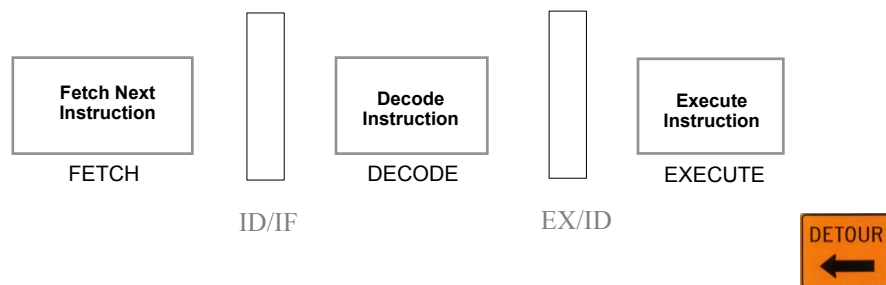
Unpipelined implementation
takes $6 \times 3 = 18$ clock cycles

Pipelined implementation takes
 $6 + 2 = 8$ clock cycles



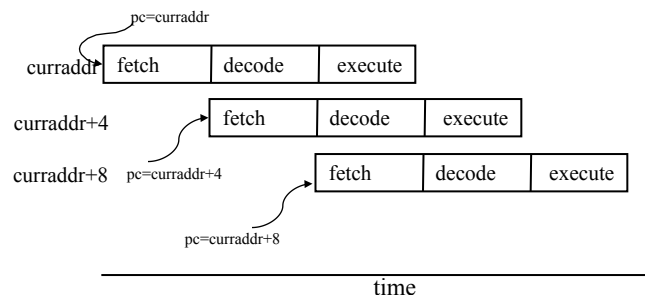
Pipelining

- Technique where multiple instructions are overlapped in execution
- Each *stage* completes a part of the execution in parallel
- Pipelining does not decrease/increase the amount of time taken for any single instruction to execute
- Pipelining increases *throughput*, i.e., number of instructions exiting the pipeline in unit time
- Pipelining has some overhead – from what?



ARM's 3-Stage Pipeline

- ARM processors (the earlier versions) employed a 3-stage pipeline with the following stages
 - Fetch: instruction fetched from memory,
 - Decode: instruction is decoded and
 - Execute: instruction is executed



- When processor is executing simple data processing instructions the pipeline enables one instruction to be completed every clock cycle
- Current versions of ARM processor have 5-stage, 6-stage or 7-stage pipeline

Effect of Prefetch on Program Counter

- Consequence of prefetch in pipelining is that the `pc` must point two instructions (8 bytes) ahead of the current instruction during the Execute cycle (when registers are read)
- Typically, the assembler or compiler handles all the detail
- On the other hand, if the programmer explicitly loads values into `pc`, he/she has to account for the prefetch in computing the value
- ARM processors might have more than 3 stages in their pipelines
 - Effect of prefetch on `pc` remains unchanged – still an 8-byte offset
- Summary
 - When an instruction is executing, if the instruction uses `pc` as a register, the value of `pc` will be the **curraddr (address of the current instruction) + 8**
 - If the **value of `pc` is not changed** by the instruction, then the next instruction to execute will be the instruction at address `curraddr+4`
 - If the **value of `pc` is changed** by the instruction, the next instruction to execute will be the instruction at address equal to the value of `pc`



More on Pipelining – Number of Stages

- Processors can vary in the number of pipeline stages
- Advantage of increasing the number of stages – each stage does less work and therefore the clock frequency can be increased further
- Example – 5 stage pipeline
 - Fetch instructions from memory (F)
 - Decode the instruction and read the registers from register file (D)
 - Execute the instruction or calculate an address (E)
 - Access an operand in data memory (M)
 - Write the result into a register (WB)
 - If all stages are balanced, ideally each stage can finish in one fifth of the time of a non pipelined implementation
- Often some combination or extension of the above steps

Issues in Pipelining

- Structural hazards – pipelining was possible because different functional units (fetch, decode, execute) were not used for an instruction at the same time
- Example

Clk	1	2	3	4	5	6	7	8
ADD r0, r1, r2	F	D	E	M	WB			
SUB r3, r0, r2		F	D	E	M	WB		
LDR r4, [r3, #4]			F	D	E	M	WB	
SUBS r5, r4, r0				F	D	E	M	WB
BEQ label					F	D	E	M
ADD r6, r7, r8						F	D	E

Both operations need to use the memory bus in the same clock cycle

Issues in Pipelining (2)

- One option – stall the pipeline (called introducing bubbles in the pipeline)

Clk	1	2	3	4	5	6	7	8
ADD r0, r1, r2	F	D	E	M	WB			
SUB r3, r0, r2		F	D	E	M	WB		
LDR r4, [r3, #4]			F	D	E	M	WB	
SUBS r5, r4, r0				F	D	E	M	WB
BEQ label					F	D	E	M
ADD r6, r7, r8							F	D

No fetch in this clock cycle because memory bus used by LDR

More Issues in Pipelining

- Data hazards

Clk	1	2	3	4	5	6	7	8
ADD r0, r1, r2	F	D	E	M	WB			
SUB r3, r0, r2		F	D	E	M	WB		
LDR r4, [r3, #4]			F	D	E	M	WB	
SUBS r5, r4, r0				F	D	E	M	WB
BEQ label					F	D	E	M
ADD r6, r7, r8						F	D	E

Value of r0 needs to be read in clock cycle 3 but available only in clock cycle 5

Data Hazards

- One option – stall the pipeline



Clk	1	2	3	4	5	6	7	8
ADD r0, r1, r2	F	D	E	M	WB			
SUB r3, r0, r2		F			D	E	M	WB
LDR r4, [r3, #4]					F	D	E	M
...								
...								

- Another option – Data forwarding

Clk	1	2	3	4	5	6	7	8
ADD r0, r1, r2	F	D	E	M	WB			
SUB r3, r0, r2		F	D	E	M	WB		
LDR r4, [r3, #4]			F	D	E	M	WB	
SUBS r5, r4, r0				F	D		E	M
BEQ label					F		D	E
ADD r6, r7, r8							F	D

Pipelining – Not Just for Breakfast Anymore

- **Branch instructions:** Next instruction to be executed is not the instruction that was decoded in the pipeline (often called *branch or control hazard*)
 - Pipeline needs to be stalled so that the proper instruction can be fetched and decoded
 - Degrades performance so need a smarter solution

Clk	1	2	3	4	5	6	7	8
ADD r0, r1, r2	F	D	E	M	WB			
SUB r3, r0, r2		F	D	E	M	WB		
SUBS r5, r4, r0			F	D	E	M	WB	
BEQ label				F	D	E	M	WB
ADD r6, r7, r8					F			
ADDS r7, r8, r9								
label								
ADD							F	

Pipelining – Not Just for Breakfast Anymore

- **Branch Prediction**
 - Predict if a branch will be taken
 - Flush pipeline if prediction is wrong

Clk	1	2	3	4	5	6	7	8
ADD r0, r1, r2	F	D	E	M	WB			
SUB r3, r0, r2		F	D	E	M	WB		
BNE label				F	D	E	M	WB
ADD r6, r7, r8					F	D	E	M
ADDS r7, r8, r9						F	D	E
label								
ADD								

More issues in Pipelining

Critical Thinking

– Okay, so what happens now if I have exceptions?

Clk	1	2	3	4	5	6	7	8
ADD r0, r1, r2	F	D	E	M	WB			
SUB r3, r0, r2		F	D	E	M	WB		
LDR r4, [r3, #4]			F	D	E	M	WB	
SUBEQNE r5, r4, r0				F	D	E	M	WB
BEQ label					F	D	E	M
ADD r6, r7, r8						F	D	E

- Precise exceptions: If the pipeline can be stopped so that all instructions before the faulting instructions can be completed while those after the faulting instruction can be stopped



ARM is a RISC Architecture

- ARM conforms to the Reduced Instruction Set Computer (RISC) architecture
- Typical of RISC systems
 - A large set of general-purpose registers that can hold either data or an address
 - Registers act as the fast local memory for all data processing operations
 - Fixed-length instructions that enable pipelining
 - Load/Store model for data processing
 - Operations on registers and not directly on memory
 - All data must be loaded into registers before they can be operated on
 - What's the advantage?
 - Small number of addressing modes
 - All load/store addresses determined from registers or instruction fields
- ARM is not a pure RISC architecture
 - In one way, this is its strength – it does not take the RISC concept too far
 - Why diverge from RISC? ARM was born to support embedded systems

Non-RISC Aspects

- Variable cycle execution for certain instructions
 - Some instructions (load-store-multiple) vary in the number of cycles depending on the number of registers involved
 - Performance features: Load-store-multiple can occur on sequential addresses (faster than random accesses)
 - Code density also improved since multiple register transfers are often at the start and the end of functions
- Inline barrel shifter, supporting more complex operations
 - Barrel shifter is a hardware component that preprocesses one of the input registers before it is used by an instruction
 - Again, improves performance and density
- 16-bit Thumb instruction set
 - Permits the ARM processor to execute either 16- or 32-bit instructions
 - Can improve code density significantly over the 32-bit ARM instructions

Non-RISC Aspects

- Conditional execution
 - Instruction executed only when a specific condition has been satisfied
 - Improves performance and code density by reducing the number of branch instructions
 - Improves the flushing of pipelines
- Enhanced instructions
 - DSP (digital signal processing) instructions
 - Support fast multiplier operations
 - In some cases, an ARM processor can replace a traditional processor + DSP coprocessor and do things faster
- Auto-increment and auto-decrement addressing modes
 - Improves execution of program loops

ATPCS

- ARM-Thumb Procedure Call Standard (ATPCS)
 - Developed by ARM
 - Defines constraints on the use of registers
 - Defines argument-passing and result-return conventions
- Compiler-generated code conforms to the ATPCS notation
 - Coding standard that becomes important to bear in mind when mixing C and assembly
- Who saves the registers?
 - A calling routine must preserve the contents of `r0-r3` if it needs them again
 - A called routine must preserve the contents of `r4-r11` and must restore their values before returning, if it has used them
 - A called routine does not need to restore `r12` before returning
 - The value held in `r13` (`sp`) on exit from a function should be the same as it was on entry
 - `r13` should not be used for any other purpose
 - Register `r14` is the link register that contains the return address back from the function

Registers in ATPCS

Register	Synonym	Special	Role in the procedure call standard
r15	-	pc	Program counter.
r14	-	lr	Link register.
r13	-	sp	Stack pointer.
r12	-	ip	Intra-procedure-call scratch register.
r11	v8	-	ARM-state variable register 8.
r10	v7	sl	ARM-state variable register 7. Stack limit pointer in stack-checked variants.
r9	v6	sb	ARM-state variable register 6. Static base in RWPI variants.
r8	v5	-	ARM-state variable register 5.
r7	v4	-	Variable register 4.
r6	v3	-	Variable register 3.
r5	v2	-	Variable register 2.
r4	v1	-	Variable register 1.
r3	a4	-	Argument/result/scratch register 4.
r2	a3	-	Argument/result/scratch register 3.
r1	a2	-	Argument/result/scratch register 2.
r0	a1	-	Argument/result/scratch register 1.

GNU Assembler (*gas*)

- Used to produce assembler output
- Has multiple directives
 - `.ascii "<string">`
 - Inserts the string as data into the assembly code
 - `.align <number>`
 - Aligns the address to $2^{\text{<number>}}$ bytes
 - `.global <symbol>`
 - Gives the symbol external linkage
 - `.section <section name>`
 - Starts a new code or data section: `.rodata`, `.text`, etc
 - `.word <word1>`
 - Inserts a list of 32-bit word values as data into the assembly code

Equivalents (C to Assembly)

```
int main (void)
{
    int a = 1024;
    int c = a + 1;
    printf ("Sum is %d\n", c);
}
```



```
.file "hello.c"
.section .rodata
.align 2

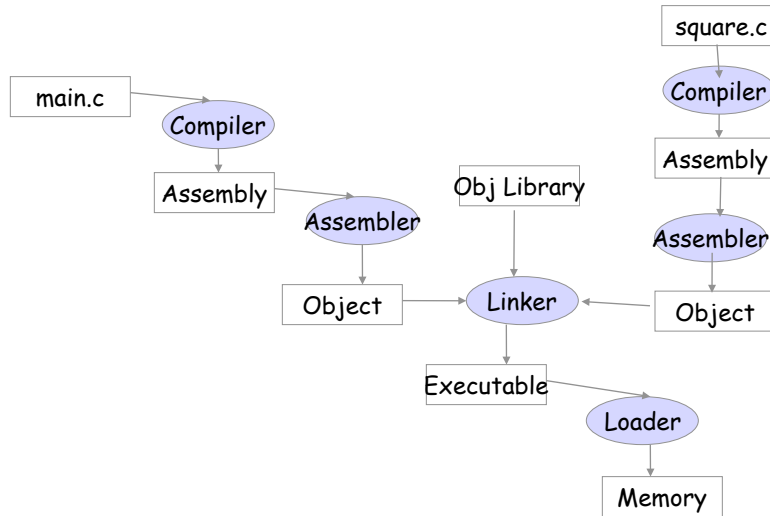
.LC0
.ascii "Sum is %d\n"
.text
.align 2
.global main

main
MOV r3, #1024
ADD r3, r3, #1
MOV r1, r3
LDR r0, .L2
BL printf

.L2
.word .LC0
```




Program Translation



Source : D. Patterson, J. Hennessey *Computer Organization & Design*



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

<pre> 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; } </pre> <p><i>main.c</i></p>	<pre> 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 </pre> <p><i>main.o</i></p>	<pre> extern int counter; int square(int x) { int result; if(counter >= 0) result=x*x; else result=0; counter--; return result; } 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 </pre> <p><i>square.c</i></p>
--	--	---

Example: After Linking main.o and square.o

<pre> 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 </pre>	<pre> 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 </pre>
---	---

linker adds the actual address of symbol *square*

linker relocates the code to a different memory location

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 1 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

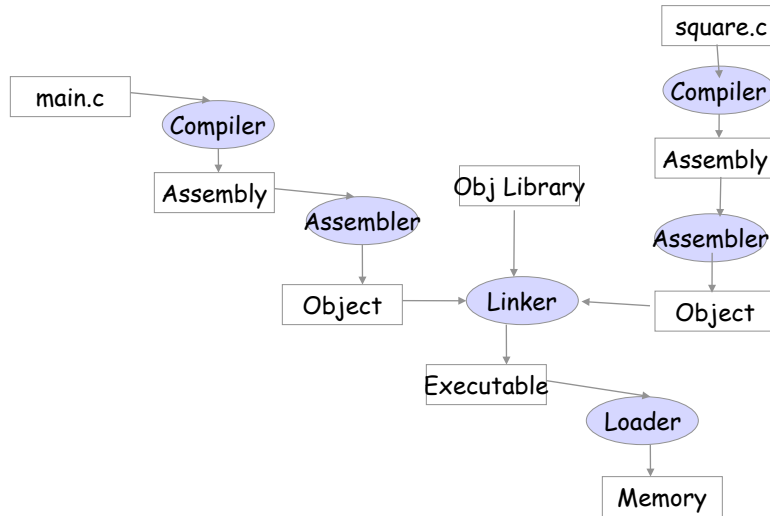
- Three main types of object files
 - **Re-locatable file**: Code and data suitable for linking with other object files to create an executable or a shared object file
 - **Executable file**: Program suitable for execution
 - **Shared object file (also called “Dynamically linked library”)**: Special 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 road map 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



Program Translation

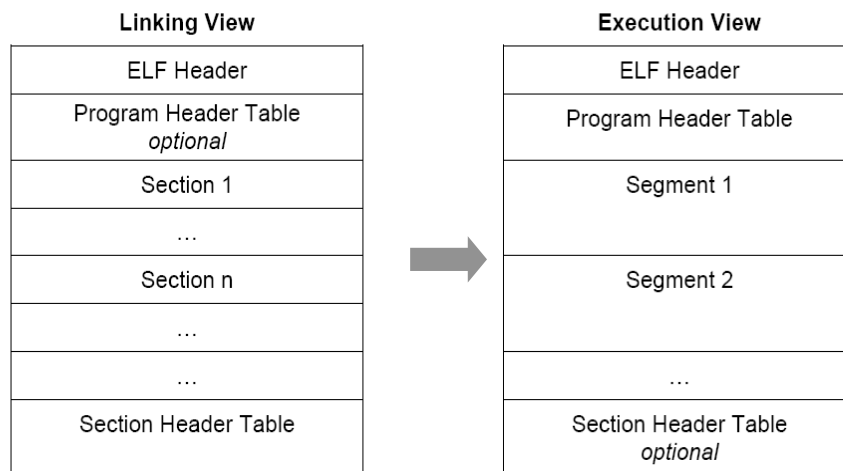


Source : D. Patterson, J. Hennessey *Computer Organization & Design*

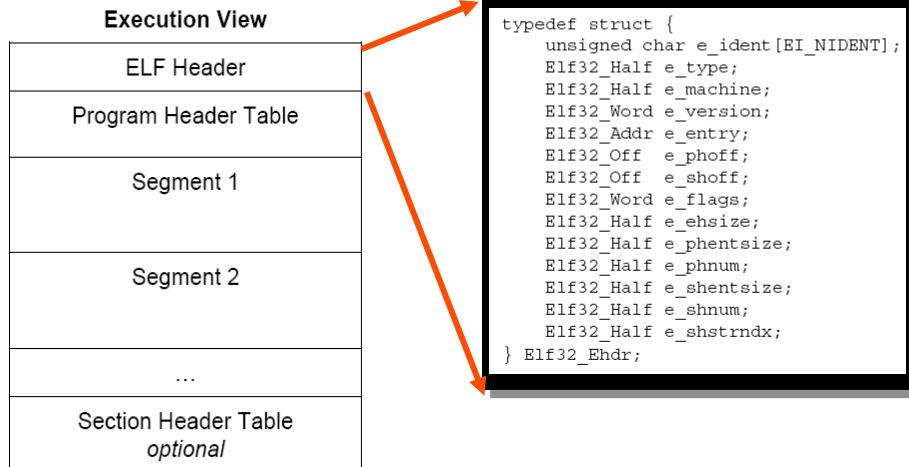
DETOUR



Linking & Execution Views



ELF Execution View



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

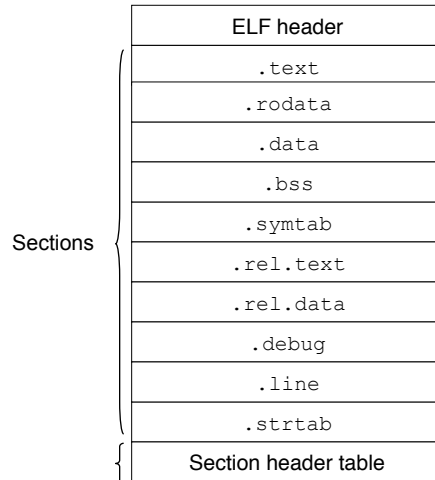
```
#define EI_NIDENT 16
typedef struct {
    unsigned char e_ident[EI_NIDENT]; // file info (object file or not)
    Elf32_Half e_type; // type of file (relocatable, executable, etc.)
    Elf32_Half e_machine; // target processor (Intel x86, ARM, SPARC etc.)
    Elf32_Word e_version; // version # (to 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

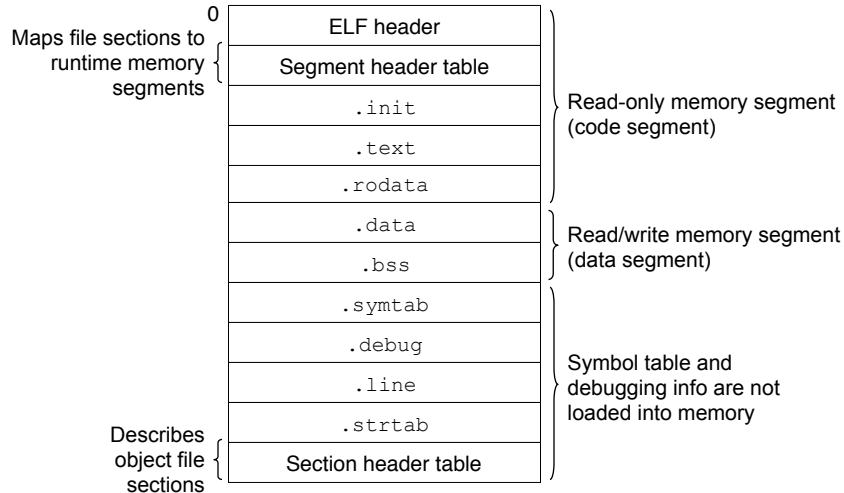


Source: "Computer Systems: A Programmer's Perspective", R. E. Bryant and D. O'Hallaron

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



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

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.o
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 ...

Summary of Lecture

- Linking
 - How linking works
 - Static & Dynamic Linking
- Loading
 - ELF file format
 - Executable & Linking Format Header
- References:
 - Chapter 7 of “*Computer Systems: A Programmer’s perspective*” by R. E. Bryant and D. R. O’Hallaron
 - Read sections 3.1, 3.2 and 3.7 of the *ARM ELF Specification* (on Blackboard in *Course Documents* ➔ *Manuals* folder)