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 micro controllers and computer architecture
- Small amount of experience with assembly

Obligatory XKCD

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…

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

```
        .syntax unified
        .arch armv7-a
        .eabi_attribute 27, 3
        .fpu vfpv3-d16
        .eabi_attribute 20, 1
        .eabi_attribute 21, 1
        .eabi_attribute 23, 3
        .eabi_attribute 24, 1
        .eabi_attribute 25, 1
        .eabi_attribute 26, 2
        .eabi_attribute 30, 6
        .eabi_attribute 18, 4
        .thumb
        .file   "hello.c"
        .section        .rodata
        .align  2
.LC0:
        .ascii  "Hello World!\000"
        .text
        .align  2
        .global main
        .thumb
        .thumb_func
        .type   main, %function
main:
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 1, uses_anonymous_args = 0
        push    {r7, lr}
        add     r7, sp, #0
        movw    r0, #:lower16:.LC0
        movt    r0, #:upper16:.LC0
        bl      puts
        mov     r3, #0
        mov     r0, r3
        pop     {r7, pc}
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.6.1-9ubuntu3) 4.6.1"
        .section        .note.GNU-stack,"",%progbits
```

And then into the following

```
Contents of section .rodata:
 83ec 01000200 48656c6c 6f20576f 726c6421  ....Hello World!
00008310 <main>:
    8310:       b508            push    {r3, lr}
    8312:       f248 30f0       movw    r0, #33776      ; 0x83f0
    8316:       f2c0 0000       movt    r0, #0
    831a:       f7ff efe0       blx     82dc <_init+0x20>
    831e:       2000            movs    r0, #0
    8320:       bd08            pop     {r3, pc}
    8322:       bf00            nop

00008324 <_start>:
    8324:       f04f 0b00       mov.w   fp, #0
    8328:       f04f 0e00       mov.w   lr, #0
    832c:       f85d 1b04       ldr.w   r1, [sp], #4
    8330:       466a            mov     r2, sp
    8332:       f84d 2d04       str.w   r2, [sp, #-4]!
    8336:       f84d 0d04       str.w   r0, [sp, #-4]!
    833a:       f8df c014       ldr.w   ip, [pc, #20]   ; 8350 <_start+0x2c>
    833e:       f84d cd04       str.w   ip, [sp, #-4]!
    8342:       4804            ldr     r0, [pc, #16]   ; (8354 <_start+0x30>)
    8344:       4b04            ldr     r3, [pc, #16]   ; (8358 <_start+0x34>)
    8346:       f7ff efd0       blx     82e8 <_init+0x2c>
    834a:       f7ff efdc       blx     8304 <_init+0x48>
    834e:       0000            .short  0x0000
    8350:       000083e1        .word   0x000083e1
    8354:       00008311        .word   0x00008311
    8358:       0000839d        .word   0x0000839d
```

Generated using objdump

Introduction to ARM

- Acorn Computers Ltd. (Cambridge, England) Nov. 1990
- 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

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

“ARM Architecture.” Wikipedia, The Free Encyclopedia. Wikimedia Foundation, Inc. 3 March 2012. Web. 3 March 2012.

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

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
R10 (SL)
r11 (FP)
r12 (IP)
r13 (SP)
r14 (LR)
r15 (PC)

CPSR

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

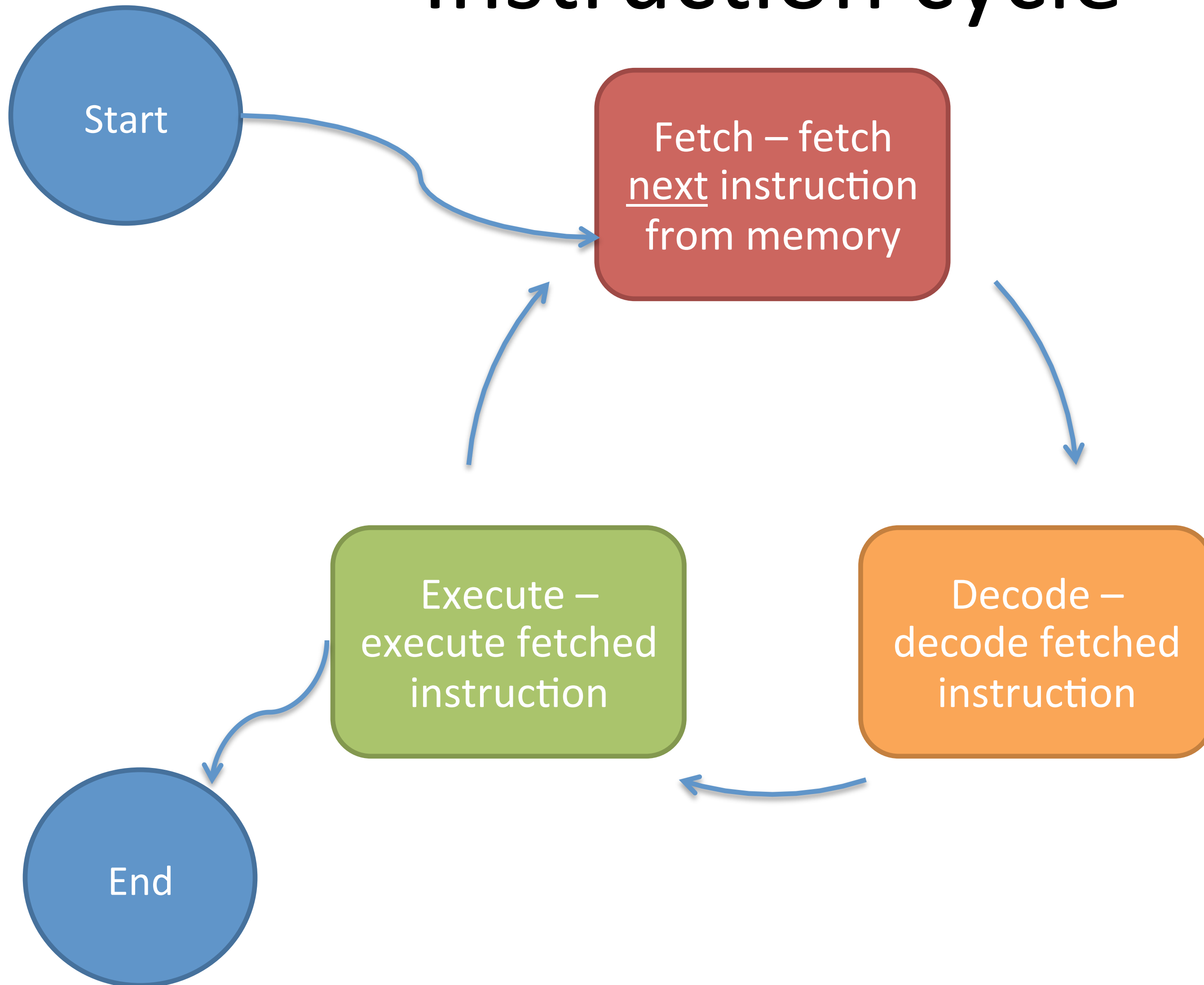
Link Register (LR) – Register used to save the PC when entering a subroutine.

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



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:
http://stackoverflow.com/questions/39419/visual-c-how-large-is-a-dword-with-32-and-64-bit-code

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

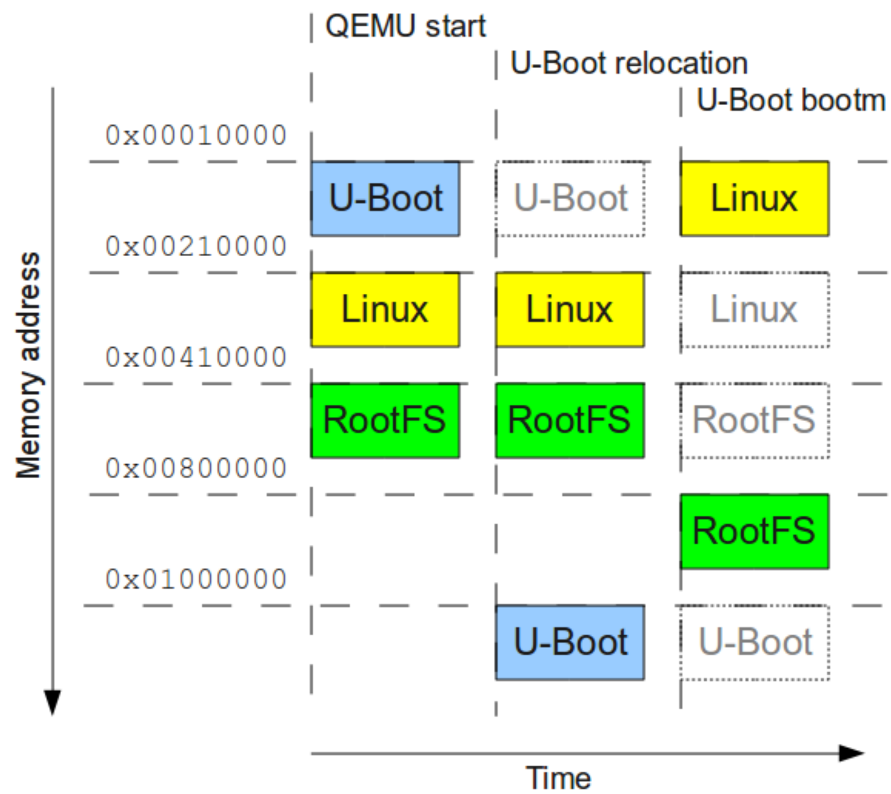
The tools we will use

- 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



Source: Balducci, Francesco.http://balau82.wordpress.com/2010/04/12/booting-linux-with-u-boot-on-qemu-arm/

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

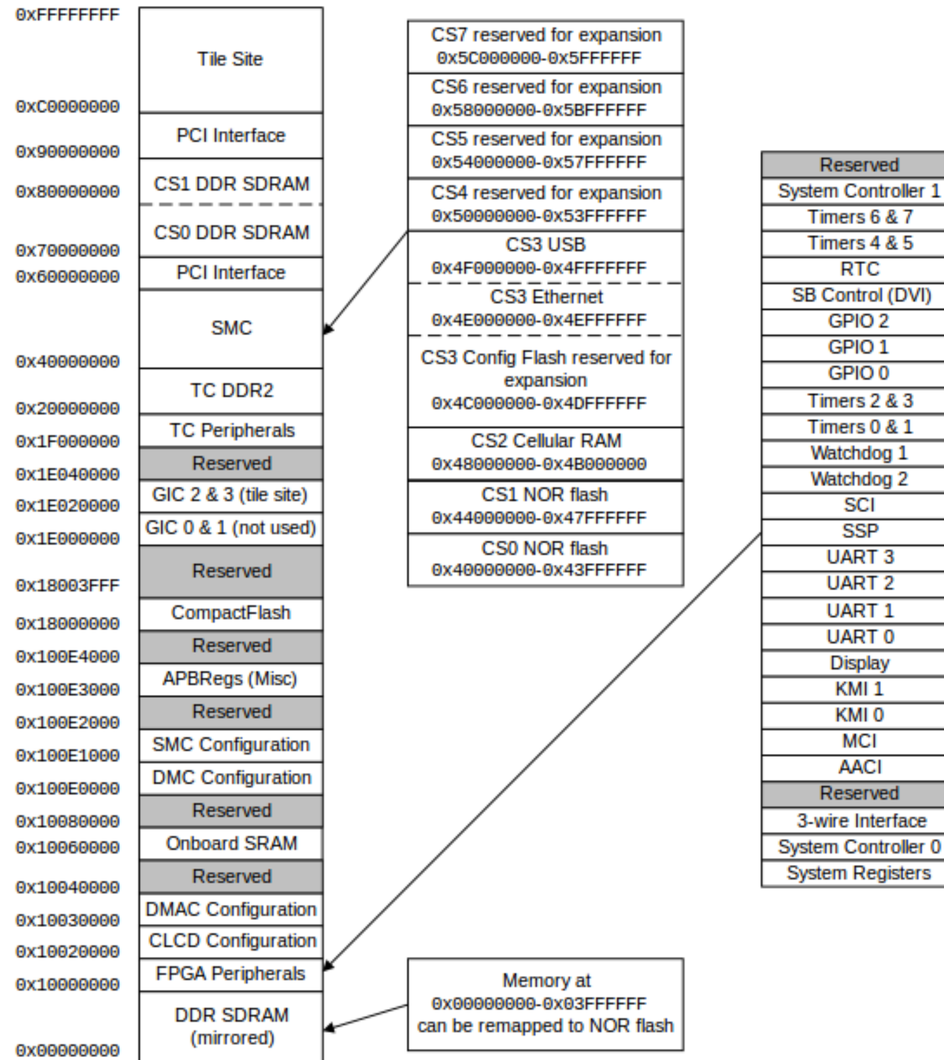
Source: Balducci, Francesco.http://balau82.wordpress.com/2010/04/12/booting-linux-with-u-boot-on-qemu-arm/

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.

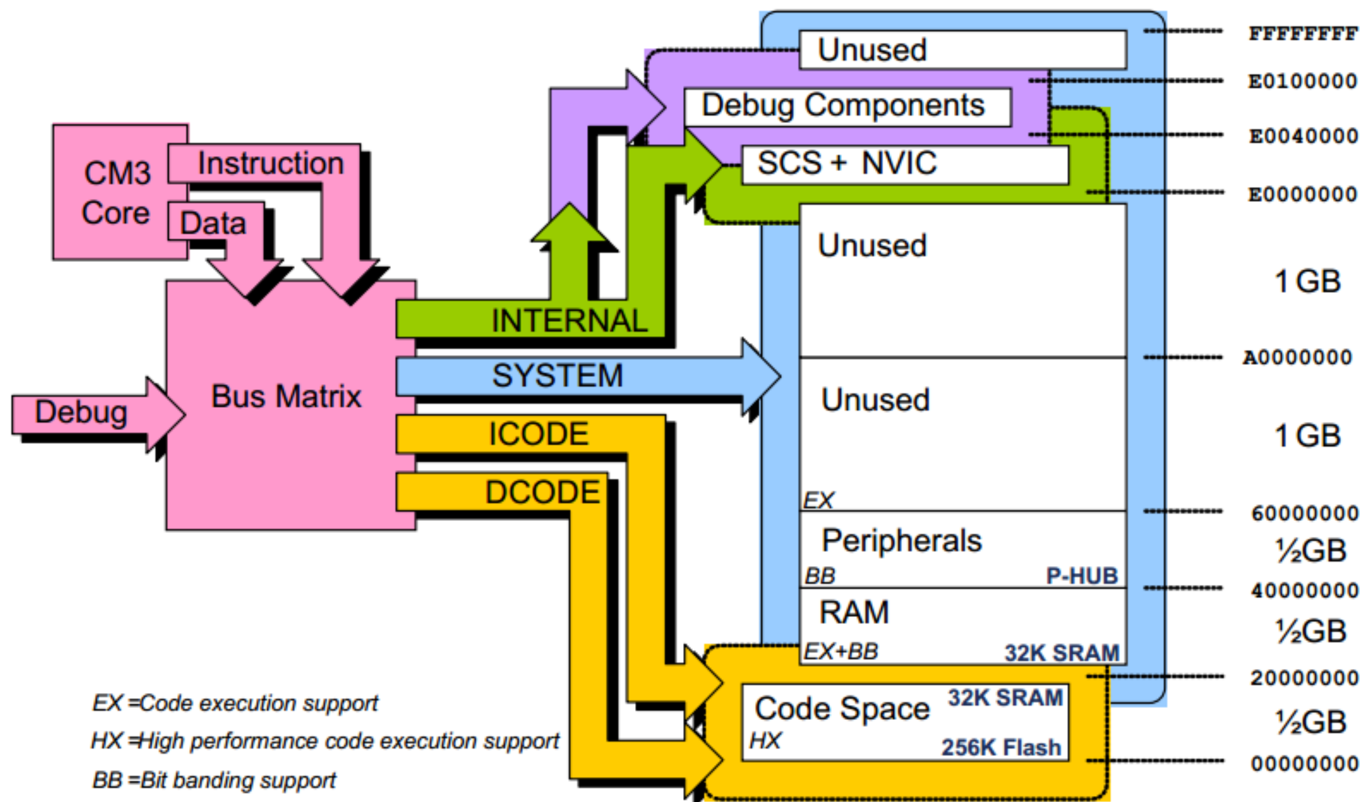
Source: Balducci, Francesco.http://balau82.wordpress.com/2010/04/12/booting-linux-with-u-boot-on-qemu-arm/

PBX w/ Cortex-A9 Memory Map



Source:
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0440b/Bbajihec.html

Cortex M3 Memory Map



Source: http://www.joral.ca/blog/wp-content/uploads/2009/10/CortexPrimer.pdf

ARM Architecture

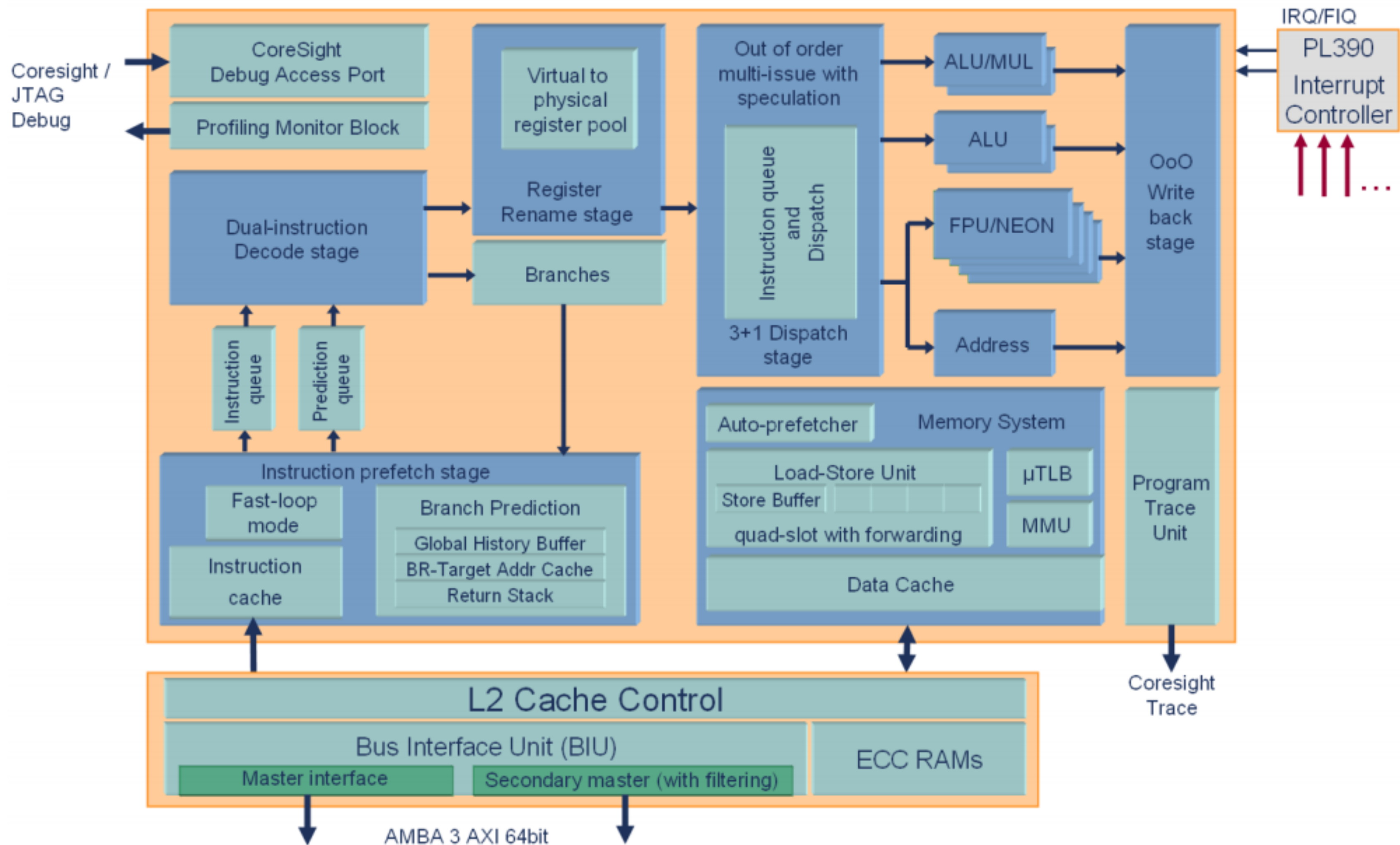
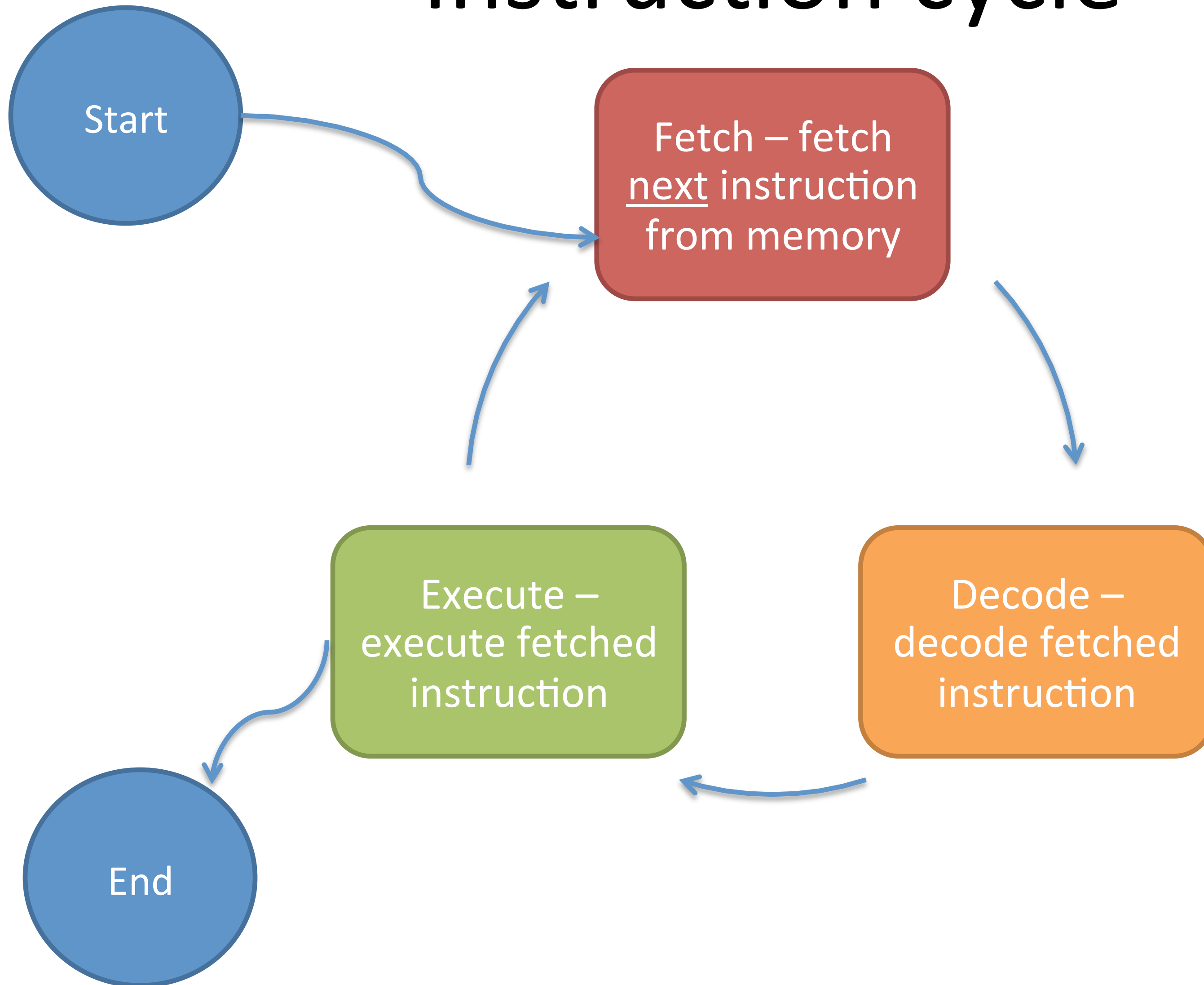


Fig. 1 Cortex-A9 microarchitecture structure and the single core interfaces.

Source: http://www.arm.com/files/pdf/armcortexa-9processors.pdf

Instruction cycle



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…

When executing instruction @ x8382 → 8382

PC=0x00008386 → 8386

```
00008380 <add>:
  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
```

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):

<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

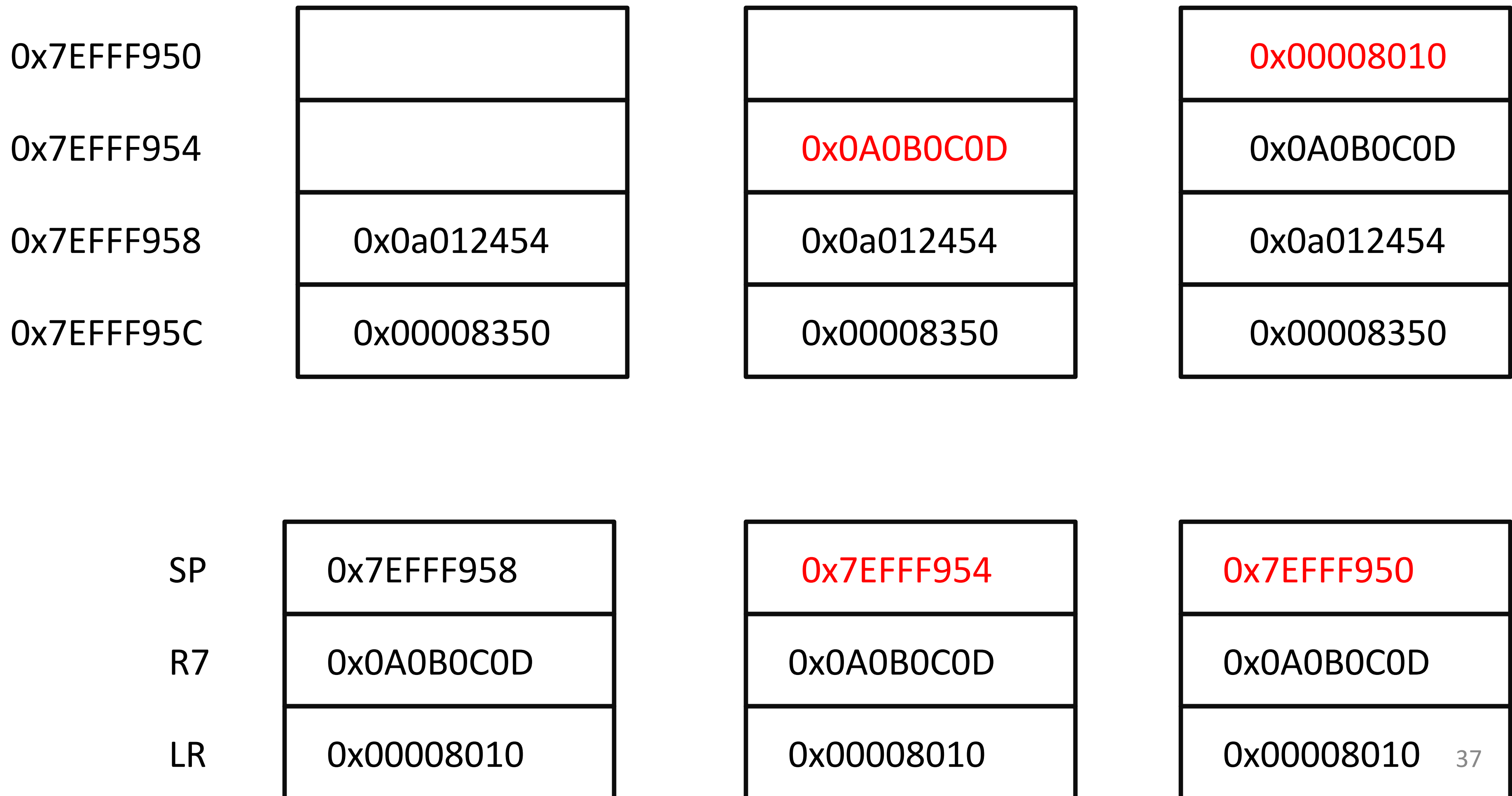
Current/Application Program Status Register (CPSR/APSR)

- **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



Arithmetic operations

- ADD: add
  - <dst> = <src> + <imm> or <src> + <reg>
- ADC: add with carry
  - <dst> = <src|c> + <imm> or <src|c> + <reg>
- SUB: subtract
  - <dst> = <src> - <imm> or <src> - <reg>
- SBC: subtract with carry
  - <dst> = <src|c> - <imm> or <src|c> - <reg>
- RSB: reverse subtract
  - <dst> = <imm> - <src> or <reg> - <src>
- RSC: reverse subtract with carry
  - <dst> = <imm|!c> - <src> or <reg|!c> - <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;
}

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.

```
00008354 <main>:
  8354:   b580        push    {r7, lr}
  8356:   b084        sub     sp, #16
  8358:   af00        add     r7, sp, #0
  835a:   f04f 030a   mov.w   r3, #10
  835e:   607b        str     r3, [r7, #4]
  8360:   f04f 030c   mov.w   r3, #12
  8364:   60bb        str     r3, [r7, #8]
  8366:   6878        ldr     r0, [r7, #4]
  8368:   68b9        ldr     r1, [r7, #8]
  836a:   f000 f809   bl      8380 <add>
  836e:   60f8        str     r0, [r7, #12]
  8370:   f04f 0300   mov.w   r3, #0
  8374:   4618        mov     r0, r3
  8376:   f107 0710   add.w   r7, r7, #16
  837a:   46bd        mov     sp, r7
  837c:   bd80        pop     {r7, pc}
  837e:   bf00        nop

00008380 <add>:
  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
```

SBC & RSB operations

	sbc	rsb
INSTRUCTION:	sbc r0, r0, r1	rsb r0, r0, r1
MEANS :	r0 = r0 – r1 – NOT(C)	r0 = r1 – r0 (No flags updated)

After Operation

	sbc	rsb
R0	0xF5F4F3FD	0x0A0B0C03
R1	0x0A0B0C0D	0x0A0B0C0D
CPSR	0x20000010	0x20000010

Before Operation

	sbc	rsb
R0	0x0000000A	0x0000000A
R1	0x0A0B0C0D	0x0A0B0C0D
CPSR	0x20000010	0x20000010

Arithmetic operations part 2

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

example2.c

```
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);
}
```

```
000083b8 <multiply>:
   83b8:   fb01 f000   mul.w    r0, r1, r0
   83bc:   4770        bx       lr
   83be:   bf00        nop

000083c0 <multiplyadd>:
   83c0:   fb01 2000   mla      r0, r1, r0, r2
   83c4:   4770        bx       lr
   83c6:   bf00        nop
```

MLA & MLS operations

INSTRUCTION: mla r0, r0, r1, r2
MEANS : r0 = r0 * r1 + r2

mls r0, r0, r1, r2
r0 = r2 – (r0 * r1) (No flags updated)

After Operation

	mla	mls
R0	0x0000008F	0xFFFFFF77
R1	0x0000000E	0x0000000E
R2	0x00000003	0x00000003
CPSR	0x20000010	0x20000010

Before Operation

	mla	mls
R0	0x0000000A	0x0000000A
R1	0x0000000E	0x0000000E
R2	0x00000003	0x00000003
CPSR	0x20000010	0x20000010

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      lr
   83ec: e1a00000          nop                          ; (mov
r0, r0)

000083f0 <unsigneddivide>:
   83f0:  e730f110         udiv     r0, r0, r1
   83f4:  e12fff1e  bx     lr
   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 dissassembling (get assembly code from the ELF format)

objdump usage

helloworld.c

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

objdump –d helloworld | less

```
Contents of section .rodata:
 83ec 01000200 48656c6c 6f20576f 726c6421  ....Hello World!
 83fc 00000000                             ....

00008310 <main>:
    8310:       b508            push    {r3, lr}
    8312:       f248 30f0       movw    r0, #33776      ; 0x83f0
    8316:       f2c0 0000       movt    r0, #0
    831a:       f7ff efe0       blx     82dc <_init+0x20>
    831e:       2000            movs    r0, #0
    8320:       bd08            pop     {r3, pc}
    8322:       bf00            nop

00008324 <_start>:
    8324:       f04f 0b00       mov.w   fp, #0
    8328:       f04f 0e00       mov.w   lr, #0
    832c:       f85d 1b04       ldr.w   r1, [sp], #4
    8330:       466a            mov     r2, sp
    8332:       f84d 2d04       str.w   r2, [sp, #-4]!
    8336:       f84d 0d04       str.w   r0, [sp, #-4]!
    833a:       f8df c014       ldr.w   ip, [pc, #20]   ; 8350 <_start+0x2c>
    833e:       f84d cd04       str.w   ip, [sp, #-4]!
    8342:       4804            ldr     r0, [pc, #16]   ; (8354 <_start+0x30>)
    8344:       4b04            ldr     r3, [pc, #16]   ; (8358 <_start+0x34>)
    8346:       f7ff efd0       blx     82e8 <_init+0x2c>
    834a:       f7ff efdc       blx     8304 <_init+0x48>
    834e:       0000            .short  0x0000
    8350:       000083e1        .word   0x000083e1
    8354:       00008311        .word   0x00008311
    8358:       0000839d        .word   0x0000839d
```

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 intruction
  - .W Wide, assembler must use 32-bit encoding for the instruction
- Do <u>not</u> 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.

Example 3.1.c

```
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;
}
```

```
00008318 <main>:
   8318:   b508        push    {r3, lr}
   831a:   2001        movs    r0, #1
   831c:   22c0        movs    r2, #192     ; 0xc0
   831e:   f248 4100   movw    r1, #33792   ; 0x8400
   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 <multiplybytwo>:
   83a8:   0040        lsls    r0, r0, #1
   83aa:   4770        bx      lr
```

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;
}
```

```
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

R0	0xFFFFFFF9
R1	0x0000000E
R2	0x00000003
CPSR	0xA0000010

R0	0xFFFFFFF8
R1	0x0000000E
R2	0x00000003
CPSR	0x20000010

asrs r0, r0, #1

R0	0xFFFFFFFC
R1	0x0000000E
R2	0x00000003
CPSR	0xA0000010

R0	0x00001000
R1	0x0000000E
R2	0x00000003
CPSR	0x20000010

RRX & LSL operation

INSTRUCTION: mvn r0, r0, RRX
MEANS : r0 = ~r0 >> 1

add r0, r0, r1, LSL #4
r0 = r0 + (r1 * 16) (No flags updated)

	mvn r0, r0, RRX	add r0, r0, r1, LSL #4
R0	0xFFFFFFFA	0x000000EA
R1	0x0000000E	0x0000000E
R2	0x00000003	0x00000003
CPSR	0xA0000010	0x20000010

R0	0x0000000A	0x0000000A
R1	0x0000000E	0x0000000E
R2	0x00000003	0x00000003
CPSR	0x20000010	0x20000010

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

```
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);
}
```

```
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
```

221412523*3 = 664237569 or 0x27977601

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;
}
```

```
0000838c <main>:
   838c:   b590        push    {r4, r7, lr}
   838e:   b085        sub     sp, #20
   8390:   af00        add     r7, sp, #0
   8392:   f04f 0306   mov.w   r3, #6
   8396:   60fb        str     r3, [r7, #12]
   8398:   f3ef 8400   mrs     r4, CPSR
   839c:   60bc        str     r4, [r7, #8]
   839e:   68fa        ldr     r2, [r7, #12]
   83a0:   f243 535d   movw    r3, #13661 ; 0x355d
   83a4:   f6cf 73fd   movt    r3, #65533 ; 0xfffd
   83a8:   18d3        adds    r3, r2, r3
   83aa:   607b        str     r3, [r7, #4]
   83ac:   f3ef 8400   mrs     r4, CPSR
   83b0:   603c        str     r4, [r7, #0]
   83b2:   f248 4344   movw    r3, #33860 ; 0x8444
   83b6:   f2c0 0300   movt    r3, #0
   83ba:   4618        mov     r0, r3
   83bc:   6879        ldr     r1, [r7, #4]
   ...
```

Before the subtraction operation

CPSR = 0x60000010

After the subtraction operation

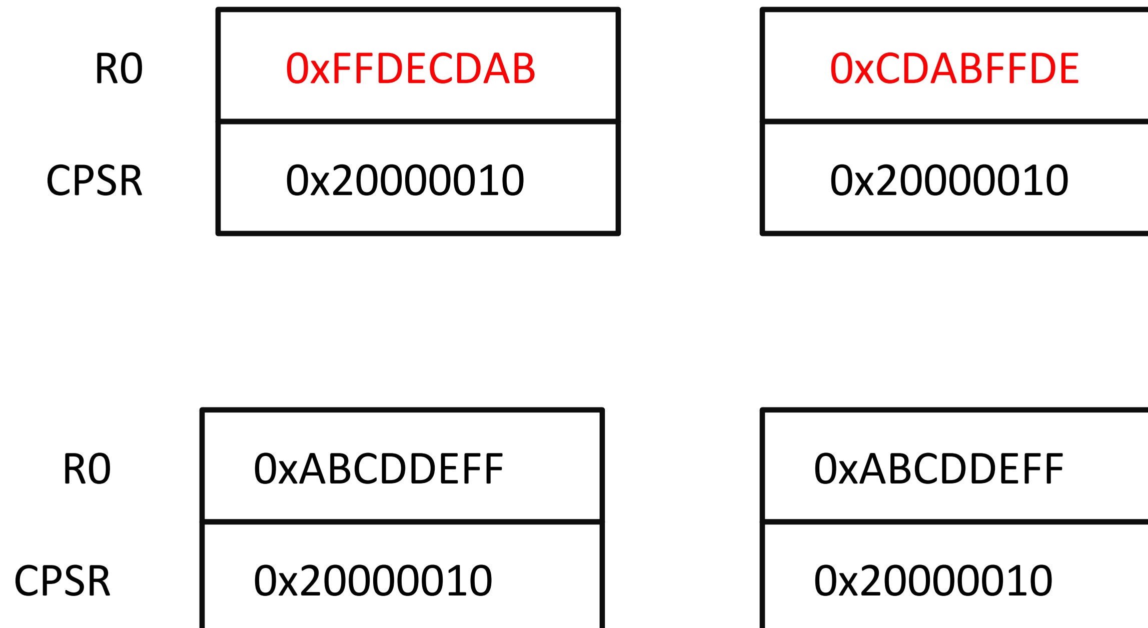
CPSR = 0x80000010

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



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**.

Example 7.1.c

```
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);
}
```

```
000083d0 <and>:
   83d0:   4008        ands      r0, r1
   83d2:   4770        bx
```

Example 7.2.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);
}
```

```
0000838c <main>:
  <prolog> ...
  8392:   f04f 0308   mov.w    r3, #8
  8396:   60bb        str      r3, [r7, #8]
  8398:   f04f 0309   mov.w    r3, #9
  839c:   607b        str      r3, [r7, #4]
  839e:   f3ef 8400   mrs      r4, CPSR
  83a2:   603c        str      r4, [r7, #0]
  83a4:   68ba        ldr      r2, [r7, #8]
  83a6:   687b        ldr      r3, [r7, #4]
  83a8:   4053        eors     r3, r2
  83aa:   2b00        cmp      r3, #0
  83ac:   dd05        ble.n    83ba <main+0x2e>
  83ae:   68b8        ldr      r0, [r7, #8]
  83b0:   6879        ldr      r1, [r7, #4]
  83b2:   f000 f829   bl       8408 <add>
  83b6:   60f8        str      r0, [r7, #12]
  83b8:   e004        b.n      83c4 <main+0x38>
  83ba:   6878        ldr      r0, [r7, #4]
  83bc:   68b9        ldr      r1, [r7, #8]
  83be:   f000 f831   bl       8424 <subtract>
  83c2:   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:    681b         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 by any one of R0-R12, SP, or LR
- We will cover consecutive multiple loads in one instruction later

Control Flow operations (Table A4-1)

Instruction	Description	Thumb mode range	ARM mode range
B <label>	Branch to target address	+/- 16 MB	+/- 32 MB
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	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!=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

```
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
```

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)

Source: http://peterdn.com/post/e28098Hello-World!e28099-in-ARM-assembly.aspx

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 intruction
  - .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 fibN(x-1) + fibN(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…

Source: http://xkcd.com/138/

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

Indexing mode	Instruction	R0	R1 or Rbase
Preindex with Writeback	LDR r0, [r1, #2]!	r0 = [r1 + 2]	r1 = r1 + 2
	LDR r0, [r1, r2]!	r0 = [r1 + r2]	r1 = r1 + r2
	LDR r0, [r1, r2, LSL #3]!	r0 = [r1 + (r2 LSL 3)]	r1 = r1 + (r2 LSL 3)
Preindex	LDR r0, [r1, #2]	r0 = [r1 + 2]	r1 = r1
	LDR r0, [r1, r2]	r0 = [r1 + r2]	r1 = r1
	LDR r0, [r1, r2, LSL #3]	r0 = [r1 + (r2 LSL 3)]	r1 = r1
Postindex	LDR r0, [r1], #2	r0 = [r1]	r1 = r1 + 2
	LDR r0, [r1], r2	r0 = [r1]	r1 = r1 + r2
	LDR r0, [r1], r2, LSL #3	r0 = [r1]	r1 = r1 + (r2 LSL 3)

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

Source: http://www.slideshare.net/guest56d1b781/arm-fundamentals

STR indexing

Indexing mode	Instruction	Rt	Rn or Rbase
Preindex with Writeback	STR r0, **[**r1, #2**]!**	[r1 + 2] = r0	r1 = r1 + 2
	STR r0, **[**r1, r2**]!**	[r1 + r2] = r0	r1 = r1 + r2
	STR r0, **[**r1, r2, LSL #3**]!**	[r1 + (r2 LSL 3)] = r0	r1 = r1 + (r2 LSL 3)
Preindex	STR r0, **[**r1, #2**]**	[r1 + 2] = r0	r1 = r1
	STR r0, **[**r1, r2**]**	[r1 + r2] = r0	r1 = r1
	STR r0, **[**r1, r2, LSL #3**]**	[r1 + (r2 LSL 3)] = r0	r1 = r1
Postindex	STR r0, **[**r1**]**, #2	[r1] = r0	r1 = r1 + 2
	STR r0, **[**r1**]**, r2	[r1] = r0	r1 = r1 + r2
	STR r0, **[**r1**]**, r2, LSL #3	[r1] = r0	r1 = r1 + (r2 LSL 3)

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

Source: 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]
(8364 <_start+0x30>)
  8354:   4b04        ldr     r3, [pc, #16]
(8368 <_start+0x34>)
  8356:   f7ff efc6   blx     82e4 <_init+0x20>
  835a:   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)

Instruction	Description	Thumb mode range	ARM mode range
B <label>	Branch to target address	+/- 16 MB	+/- 32 MB
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)

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

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**

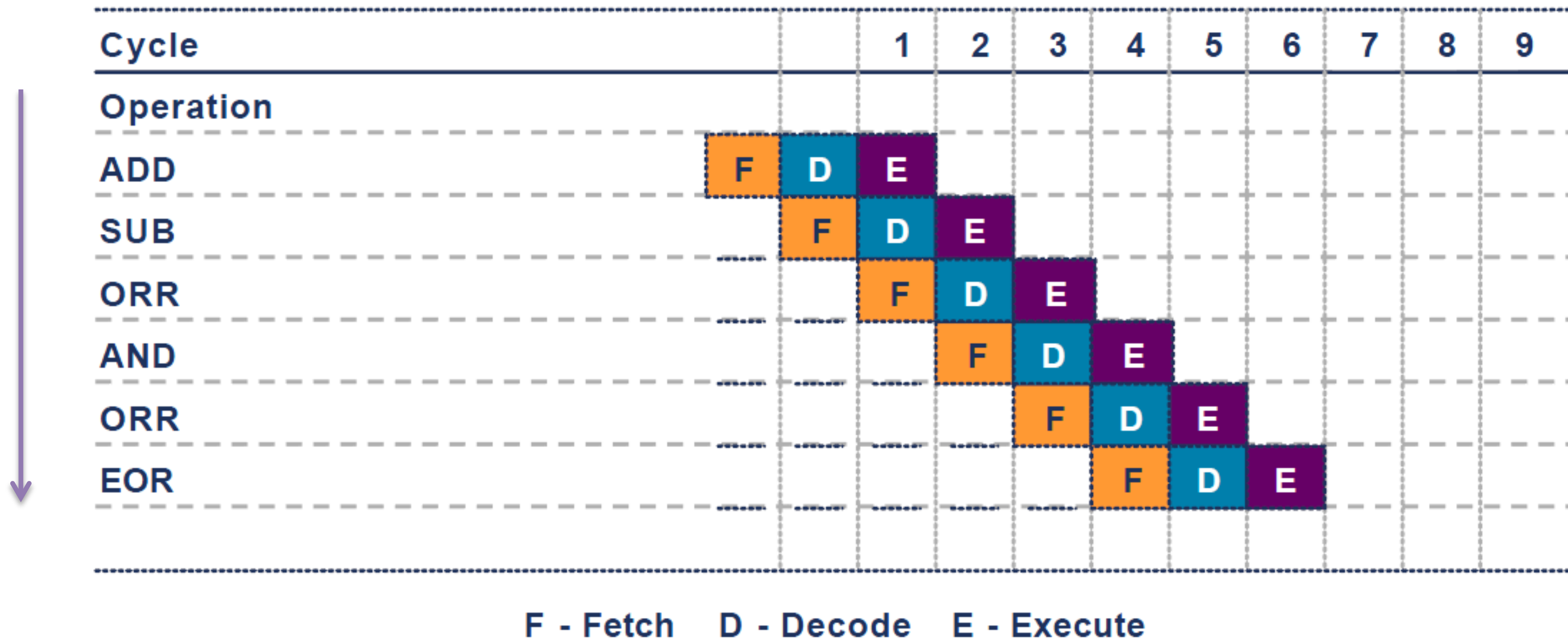
Pipelining

- 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.

Source:
http://www-cs-faculty.stanford.edu/~eroberts/courses/soco/projects/2000-01/risc/pipelining/index.html

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

Pipelining in action



Source: http://web.eecs.umich.edu/~prabal/teaching/eecs373-f10/readings/ARM_Architecture_Overview.pdf

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

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

Source: http://en.wikipedia.org/wiki/List_of_ARM_microprocessor_cores

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

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

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

Stack operations

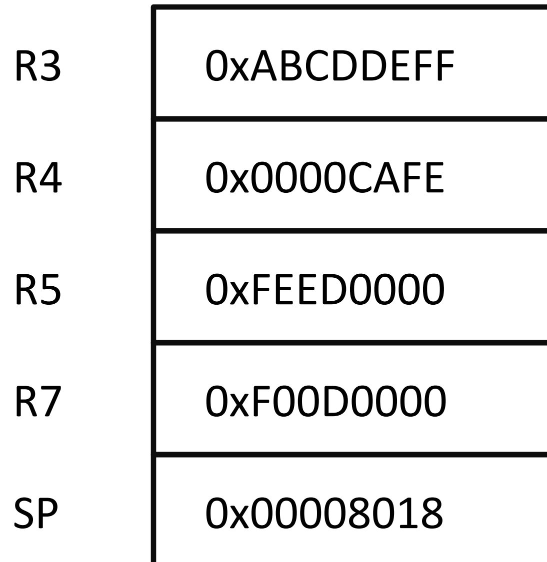
- Instead of POP, we use Load-Multiple
- Instead of PUSH, we use Store-Multiple
- Stacks can be
  - (A)scending – stack grows to higher memory addresses
  - (D)escending – stack grows to lower memory addresses

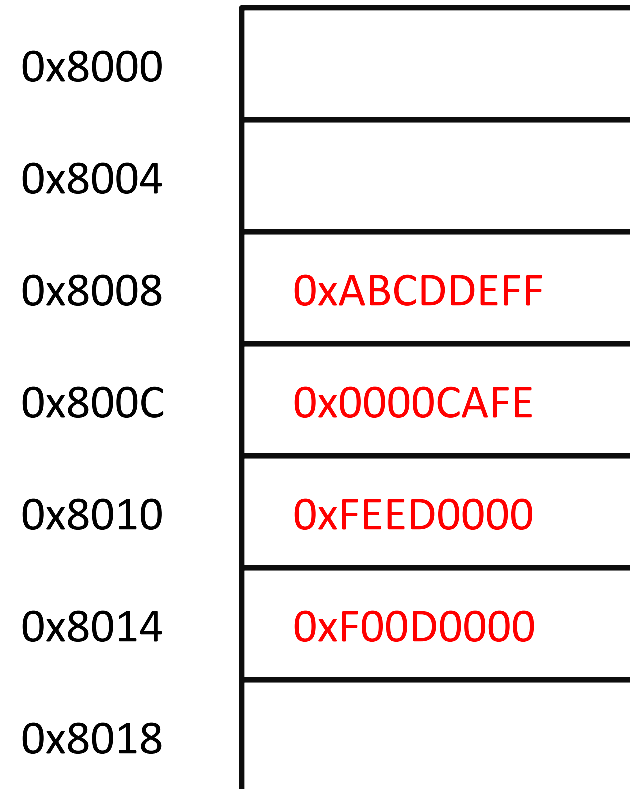
LDM/STM pairs

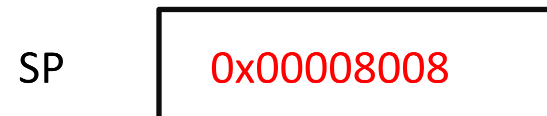
Store Multiple	Load Multiple
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

STMDB operation

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

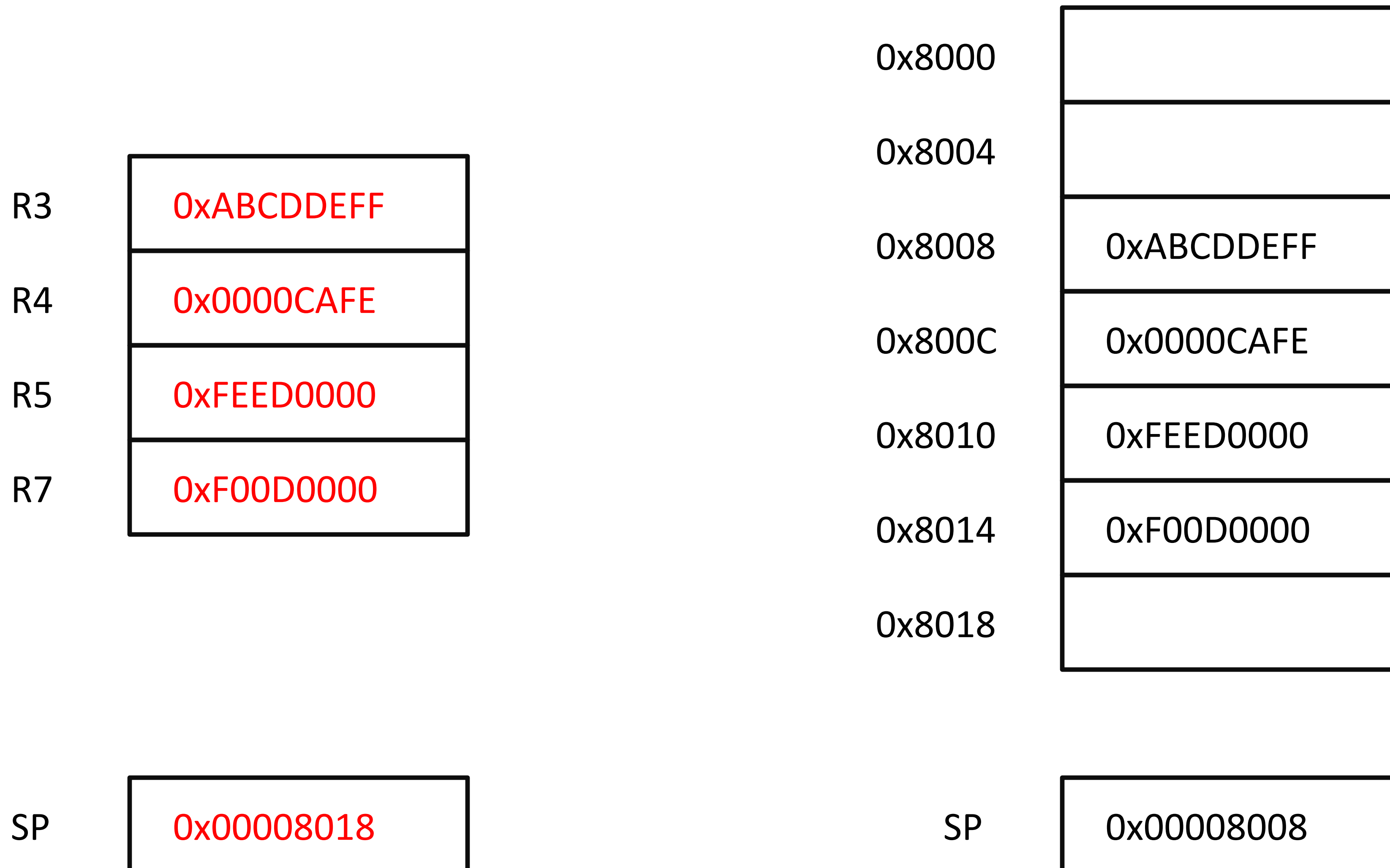






LDMIA operation

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



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

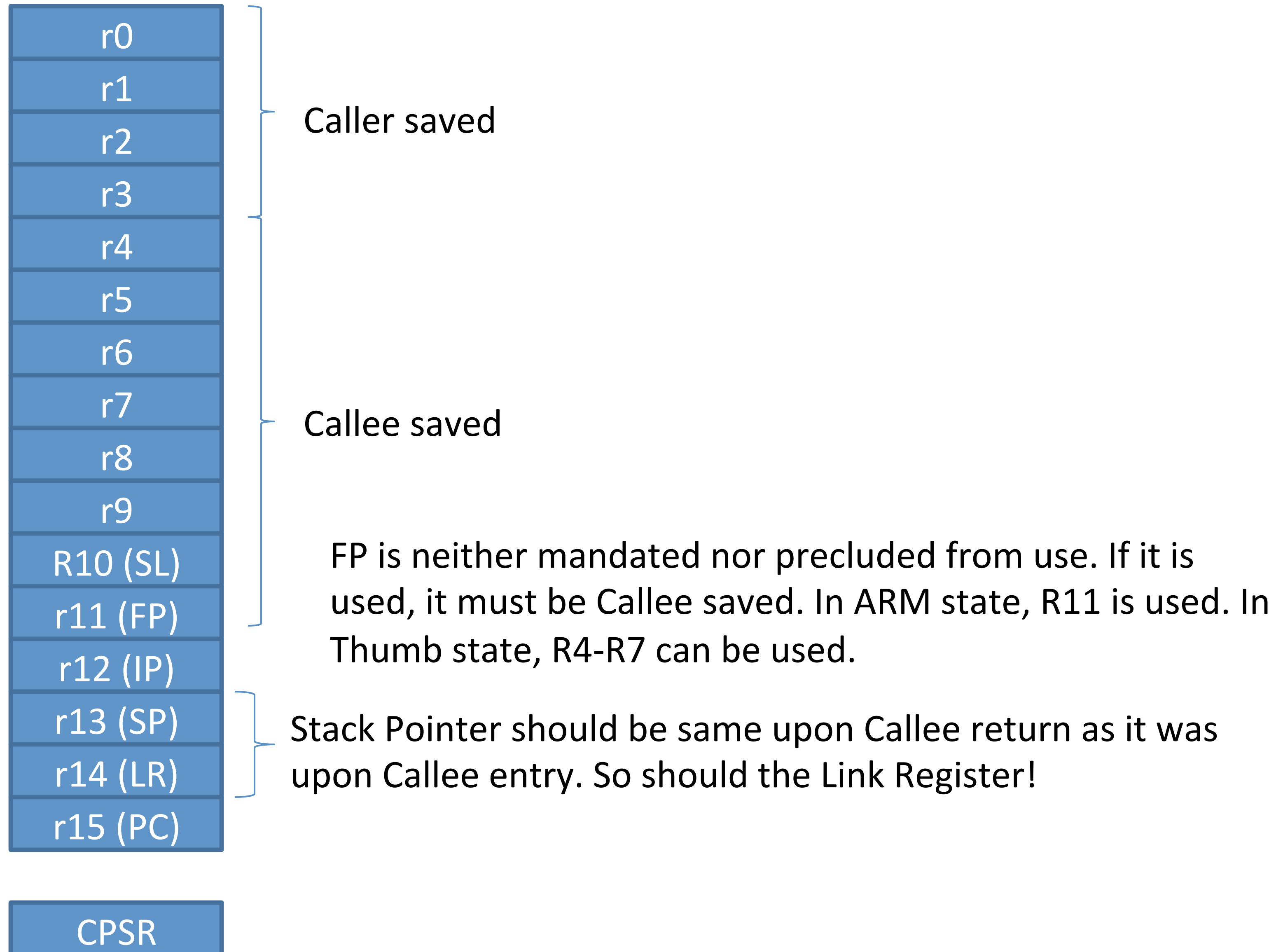
ARM-Thumb Procedure Call Standard

- 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?

ATPCS

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter. (x86 EIP)
r14		LR	The Link Register. (x86 saved EIP)
r13		SP	The Stack Pointer. (x86 ESP)
r12		IP	The Intra-Procedure-call scratch register. (x86 RSI)
r11	v8		Variable-register 8/Frame Pointer (x86 EBP)
r10	v7		Variable-register 7/Stack Limit
r9		v6/SB/TR	Platform specific register.
r8	v5		Variable-register 5.
r7	v4		Variable-register 4. (can also be x86 EBP)
r6	v3		Variable-register 3.
r5	v2		Variable-register 2.
r4	v1		Variable-register 1.
r3	a4		Argument/scratch register 4.
r2	a3		Argument/scratch register 3.
r1	a2		Argument/result/scratch register 2.
r0	a1		Argument/result/scratch register 1.

ATPCS

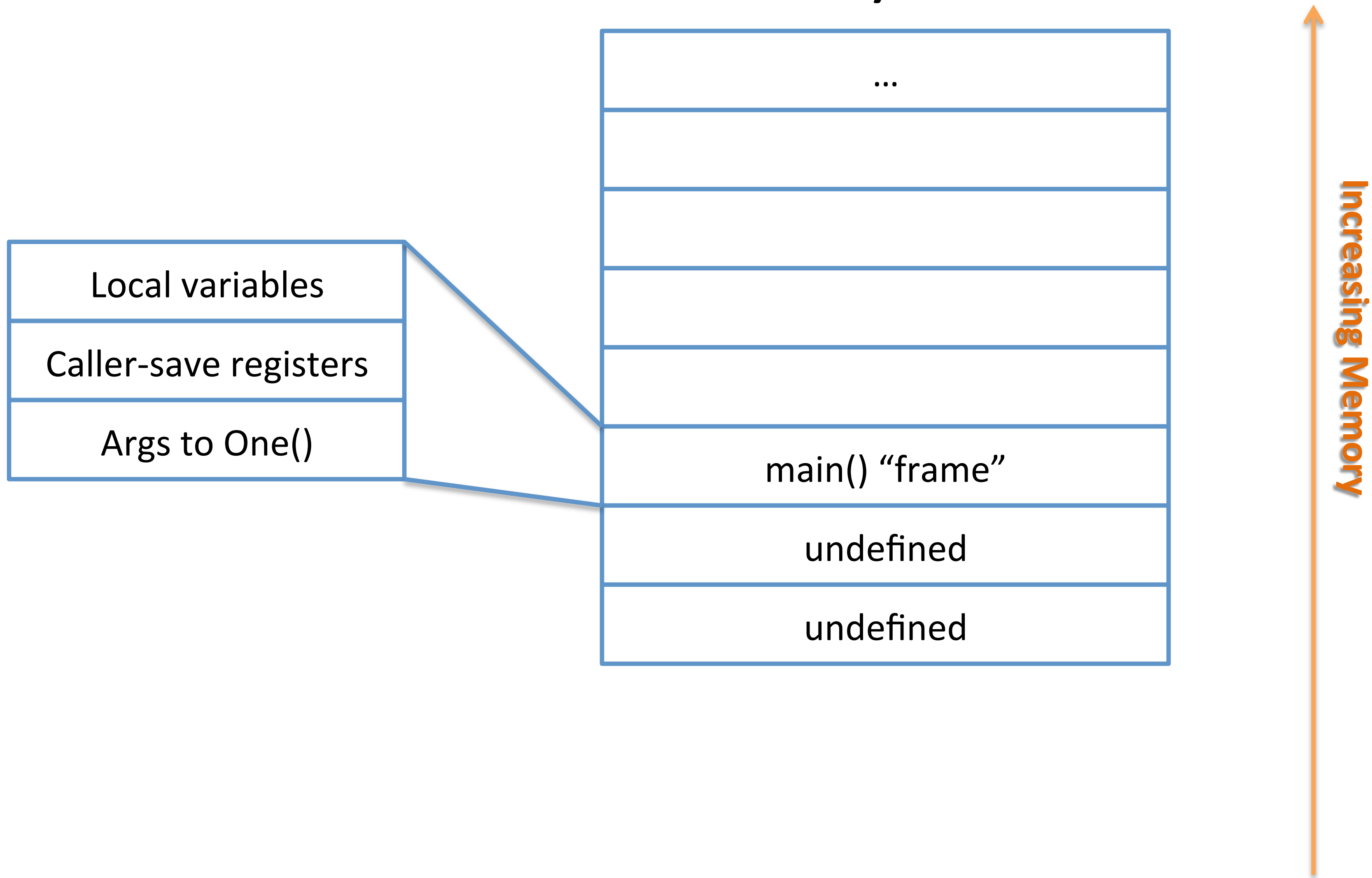


ATPCS in action

```
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;
}
```

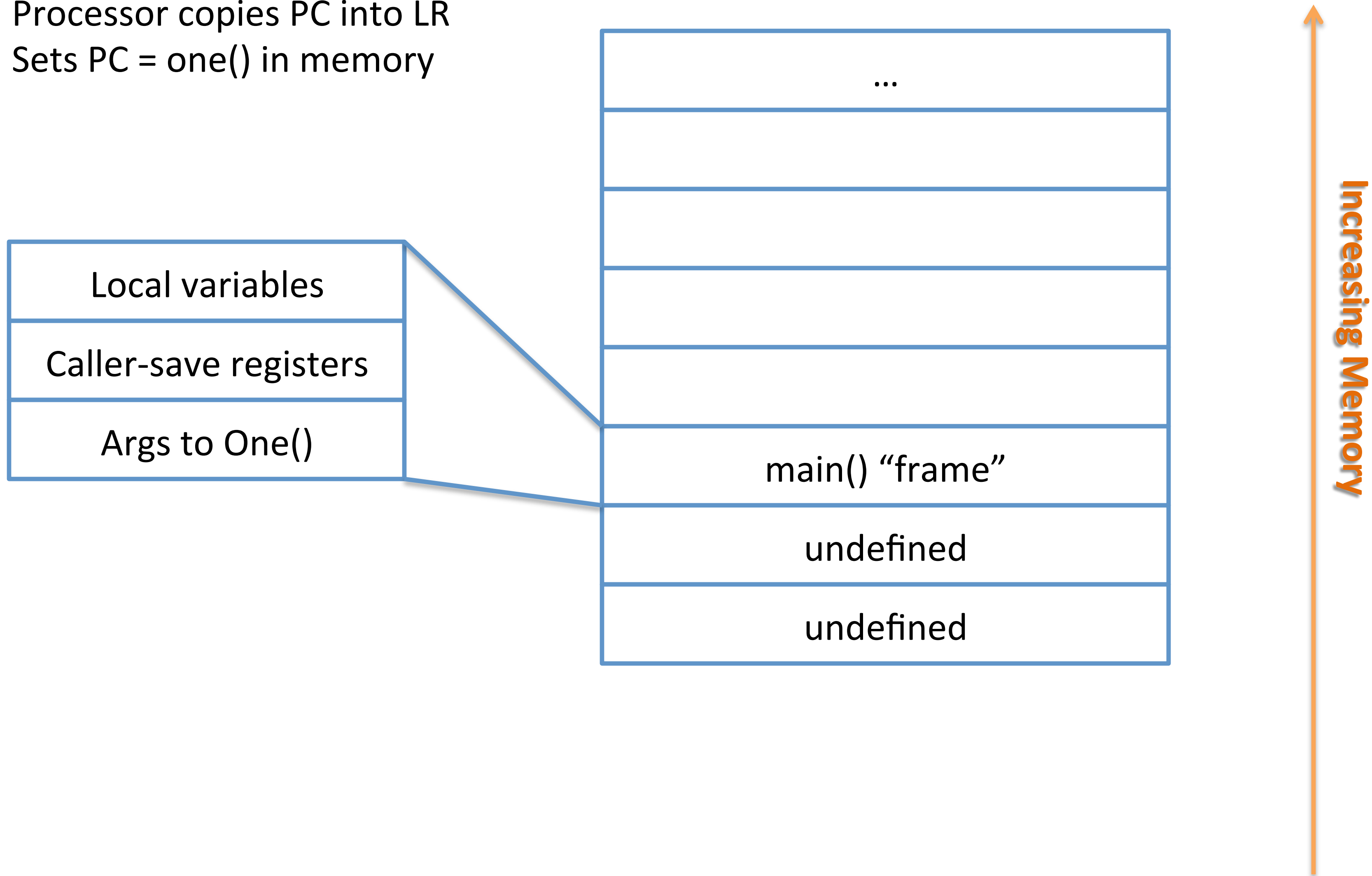
```
int main(void)
{
   r0-r3 saved.
   call to one() made.
}

void one(void)
{
   r4-r11 saved.
   lr saved
   fp saved to point to one above lr in stack
   // use r0-r3 as arguments
   two();
   r4-r11, lr restored
   bx lr (branches to lr)
}
```

So, how does this stack up? (pun intended)



Branch with Link occurs to one()

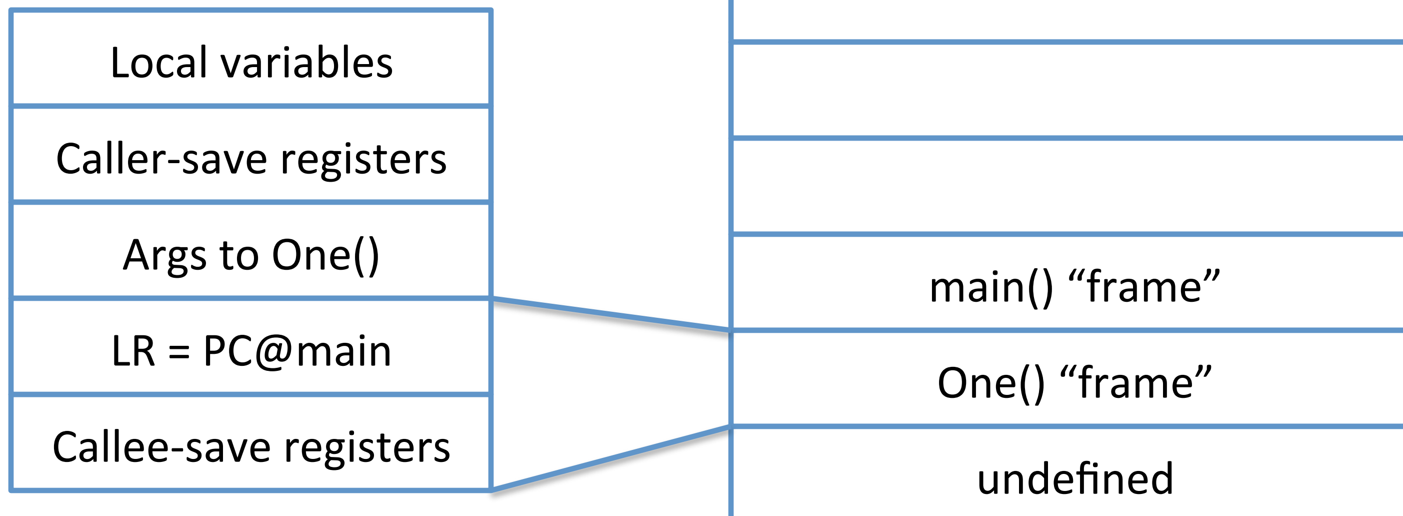
Processor copies PC into LR
Sets PC = one() in memory



ARM now executing first instruction in one()

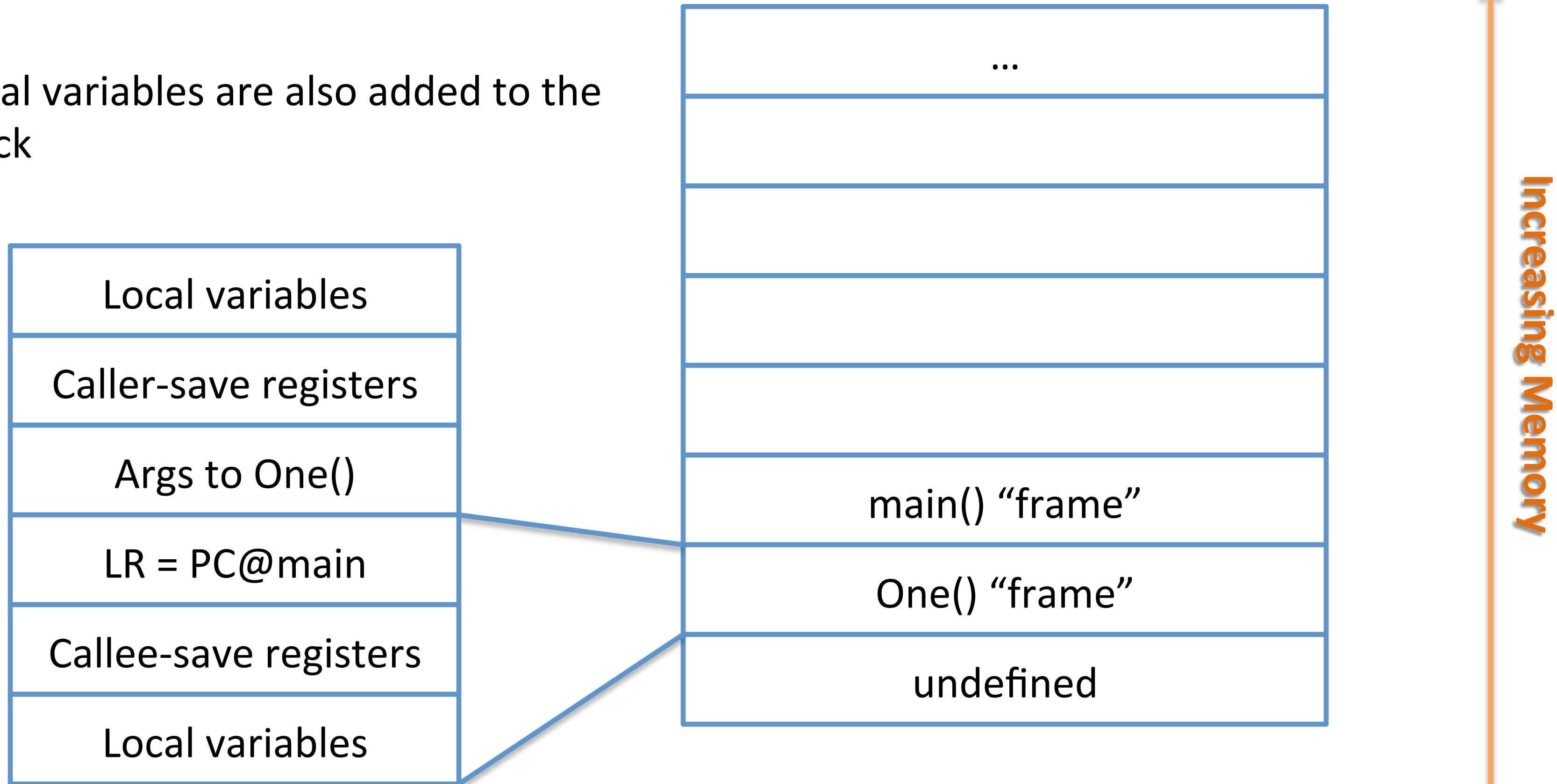
Callee-save registers are pushed onto stack using STMFD sp!, {registers} along with R14 (LR)

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

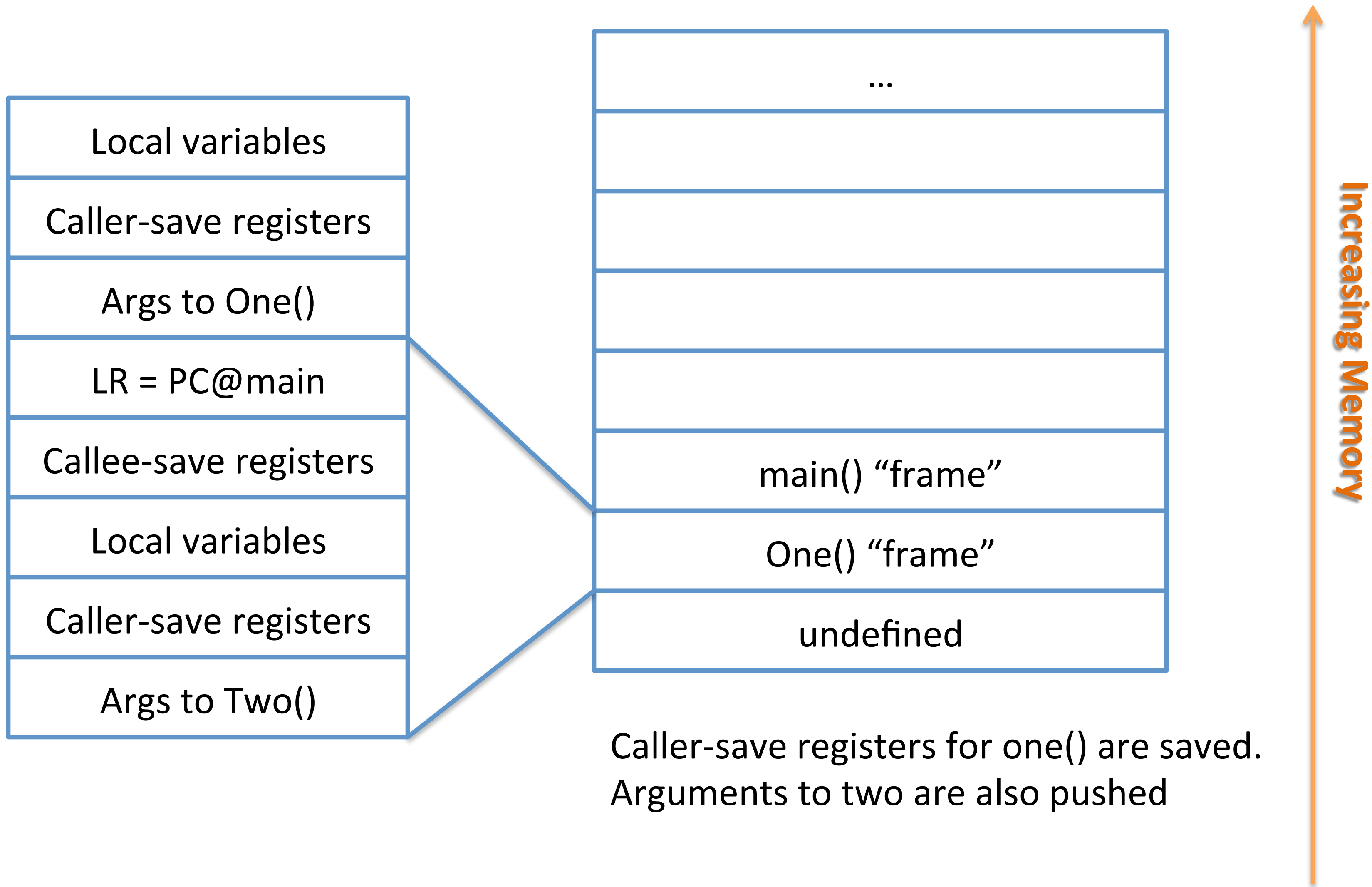


ARM now executing second instruction in one()

Local variables are also added to the stack



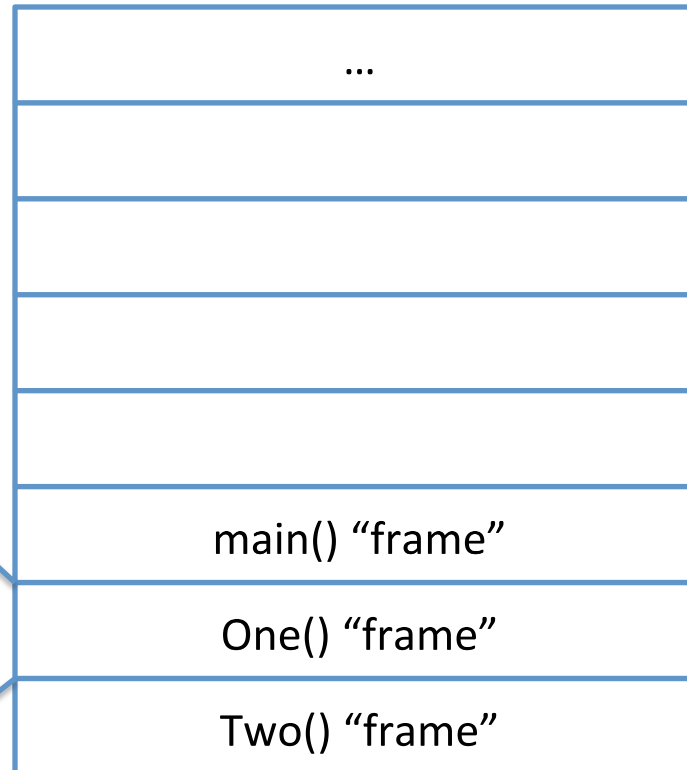
PC now about to branch to two()



Branch with Link occurs to two()

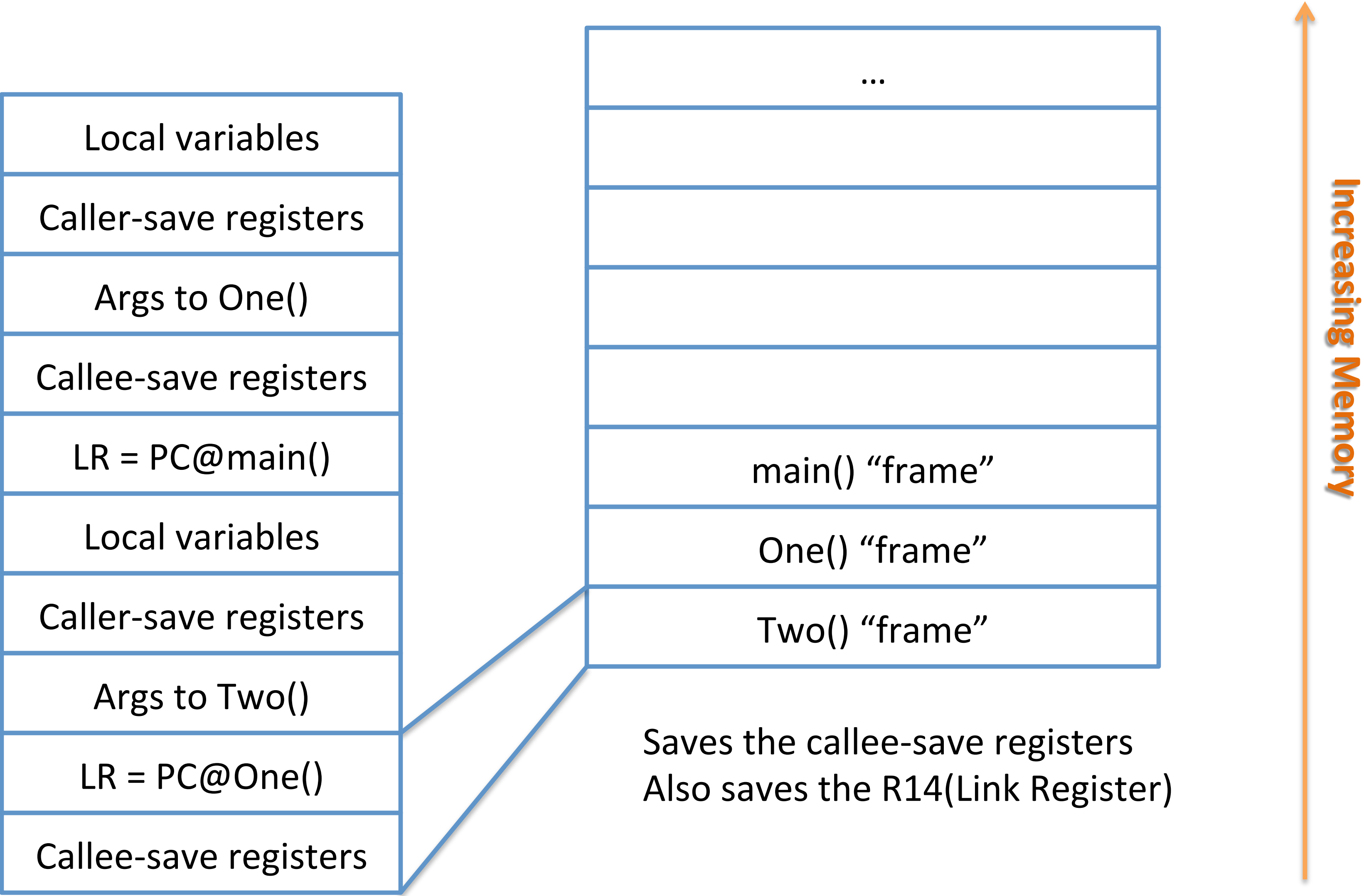
Processor copies PC into LR
Sets PC = one() in memory

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



Increasing Memory

ARM now executes first instruction in two()



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**

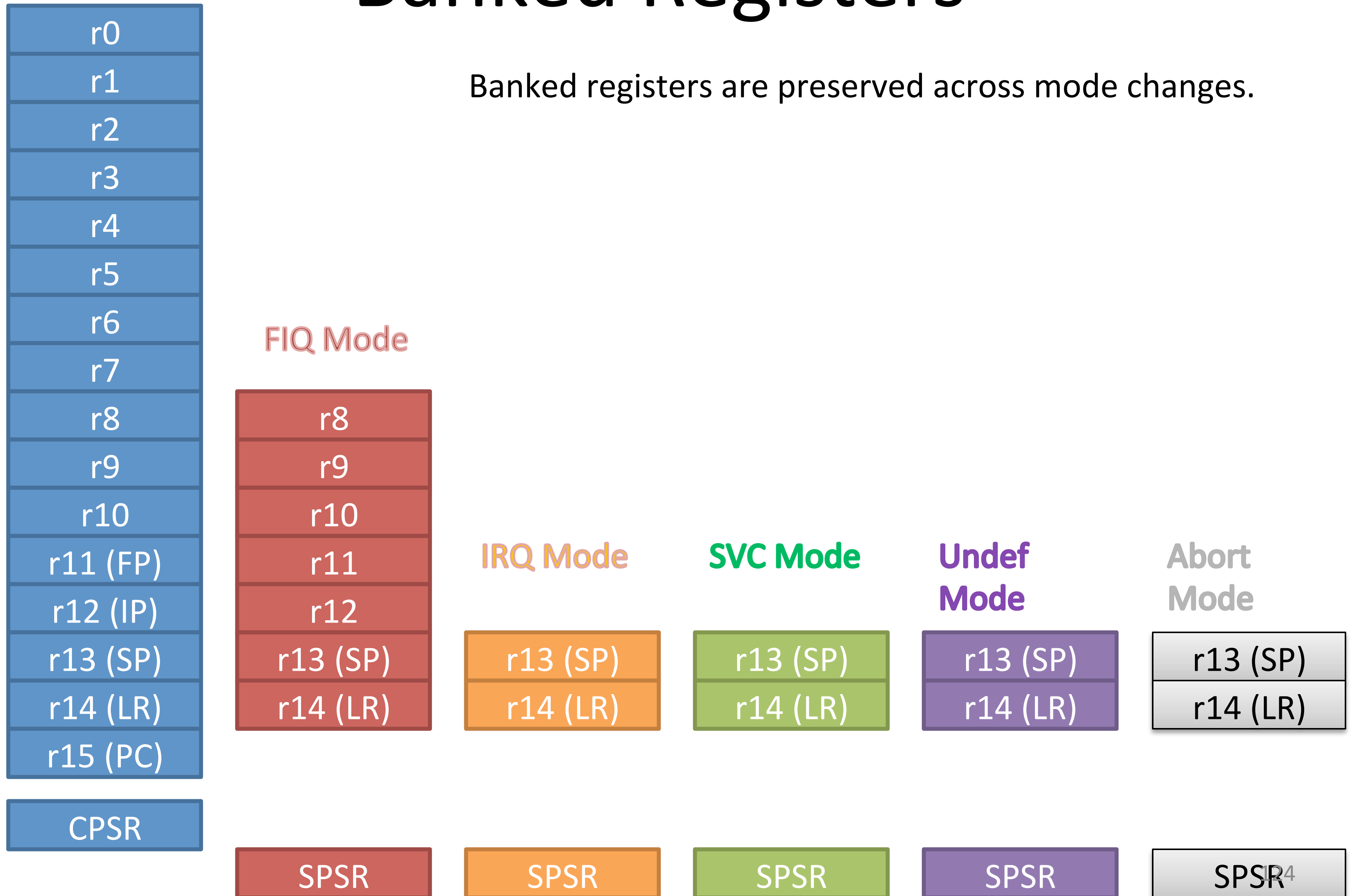
_Mode [4:0]	Mode
10000	User
10001	FIQ
10010	IRQ
10011	SVC (Supervisor)
10111	Abort
11011	Undefined
11111	System

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)

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

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 extenstions)

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 Regsiter from special register
- Example: mrs <r0> <cpsr/apsr>

SCTLR 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

SCTLR 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)

Instruction	Description	Meaning
B <label>	Branch to label	PC = &label
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 & 0xFFFFFFFE T bit = Rm & 1
BX <Rm or #imm>	Branch exchange	LR = & of instr. after BLX instr. PC = Rm & 0xFFFFFFFE T bit = Rm & 1

Source: http://www.slideshare.net/guest56d1b781/arm-fundamentals

Control Flow operations (Table A4-1)

Instruction	Description	Thumb mode range	ARM mode range
B <label>	Branch to target address	+/- 16 MB	+/- 32 MB
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

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

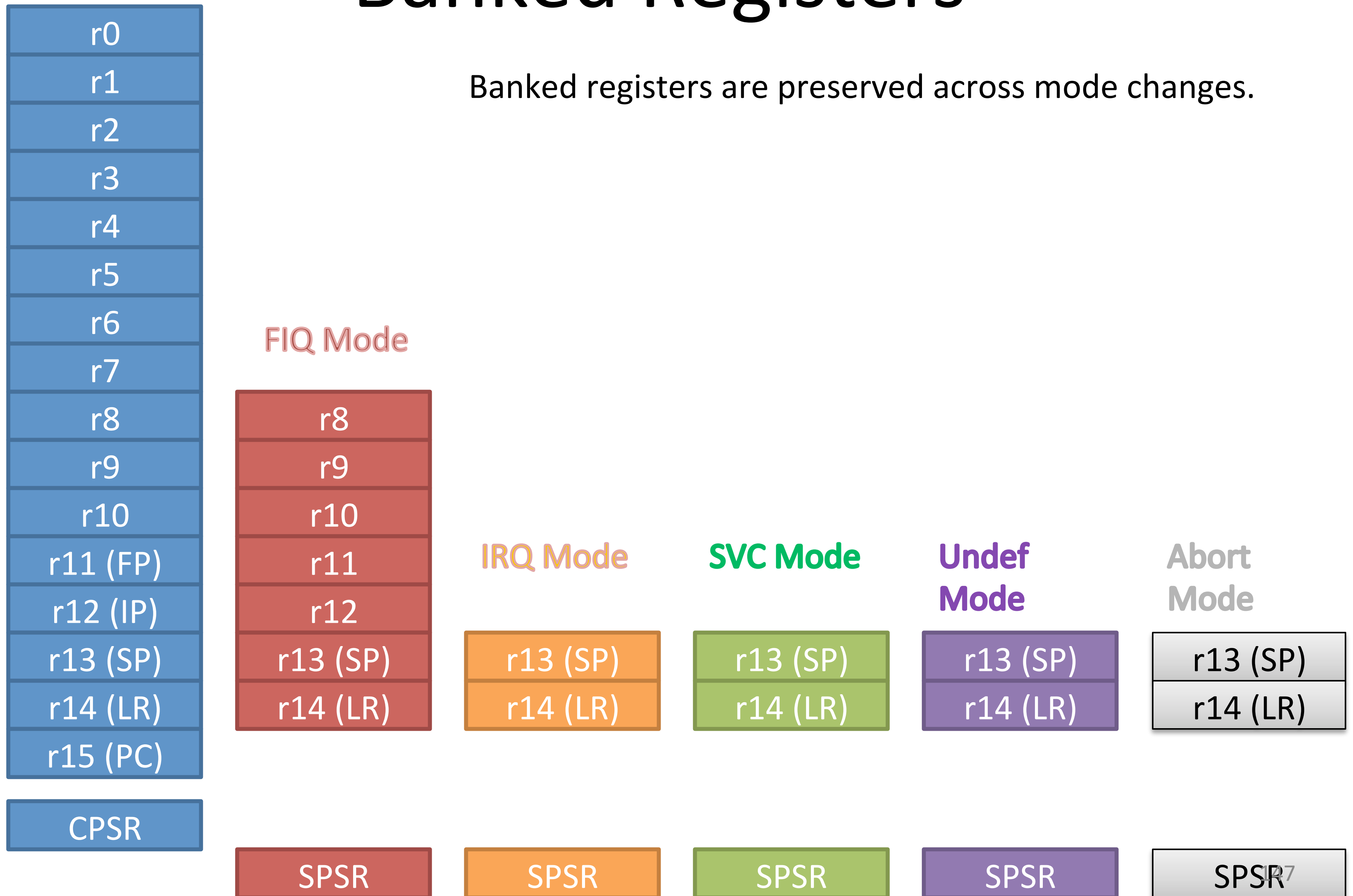
- PLD, PLDW [<reg>, <imm>] - Preload data from memory at address in <reg> with offset of <imm>
- PLI [<reg>, <imm>] – Preload instructions from memory
- DMB – Data memory barrier ensures order of memory operations
- DSB – Data Synchronization barrier ensures completion of memory access operation
- ISB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

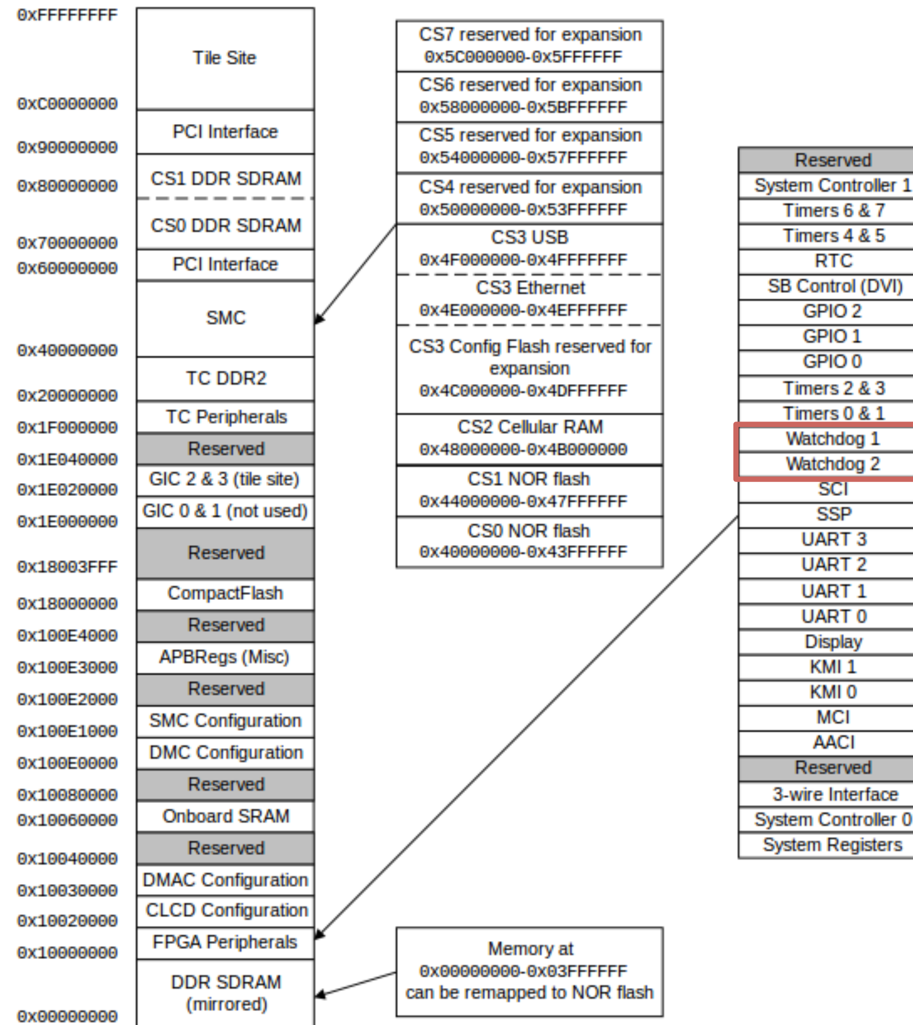


Is timing important?

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

Source: http://xkcd.com/612/

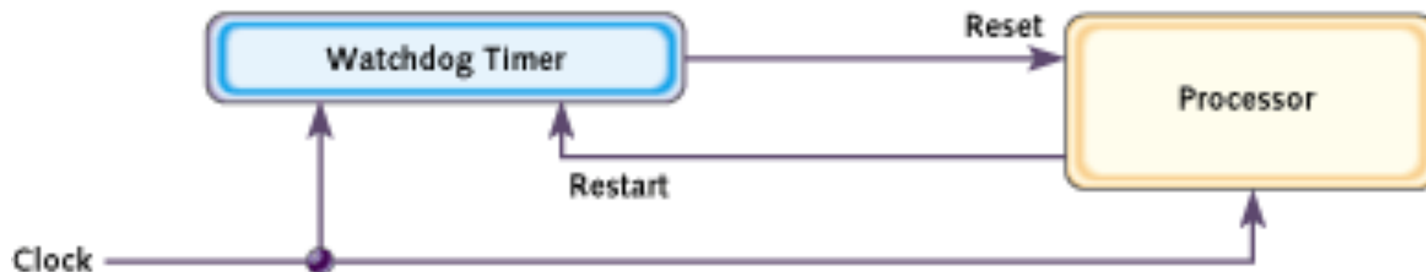
PBX-A9 Memory Map



Source: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0440b/Bbajihec.html

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



Source: http://www.eetimes.com/discussion/beginner-s-corner/4023849/Introduction-to-Watchdog-Timers

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

This week at the IEEE Real-Time Systems Symposium I heard a fascinating keynote address by David Wilner, Chief Technical Officer of Wind River Systems. Wind River makes VxWorks, the real-time embedded systems kernel that was used in the Mars Pathfinder mission. In his talk, he explained in detail the actual software problems that caused the total system resets of the Pathfinder spacecraft, how they were diagnosed, and how they were solved. I wanted to share his story with each of you.

VxWorks provides preemptive priority scheduling of threads. Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.

Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft. A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).

The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex. If an interrupt caused the information bus thread to be scheduled while this mutex was held, and if the information bus thread then attempted to acquire this same mutex in order to retrieve published data, this would cause it to block on the mutex, waiting until the meteorological thread released the mutex before it could continue. The spacecraft also contained a communications task that ran with medium priority.

Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset.

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
  - …

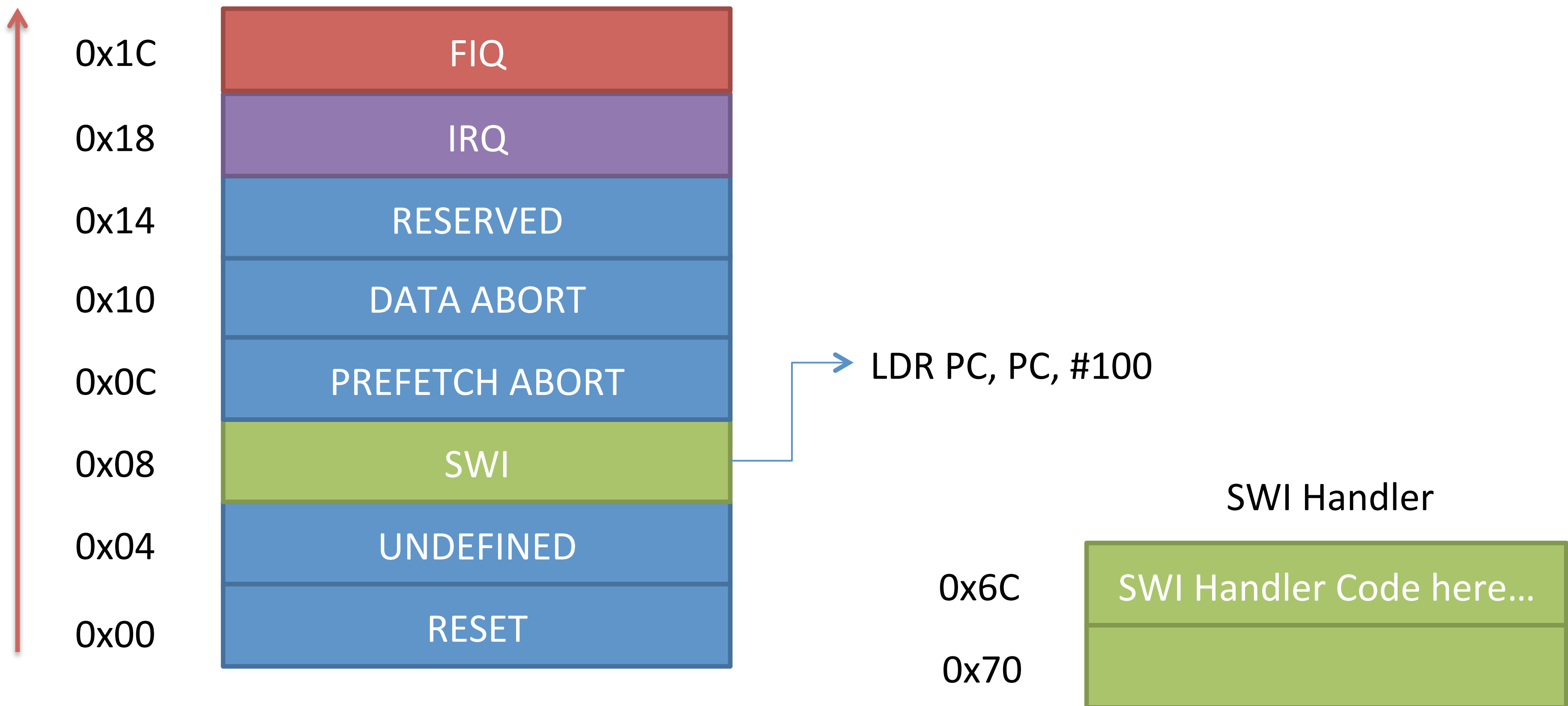
Interrupt handlers

- 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

Exception type	Mode	Vector Address	Priority
Reset	Supervisor	0x00000000	1 (highest)
Data Abort	Abort	0x00000010	2
FIQ (Fast Interrupt)	FIQ	0x0000001C	3
IRQ (Normal Interrupt)	IRQ	0x00000018	4
Prefetch Abort	Abort	0x0000000C	5
Sofware Interrupt (SWI/SVC)	Supervisor	0x00000008	6
Undefined instruction	Undefined	0x00000004	6 (lowest)

Interrupt vector table



Current Program Status Register

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

_Mode [4:0]	Mode
10000	User
10001	FIQ
10010	IRQ
10011	SVC (Supervisor)
10111	Abort
11011	Undefined
11111	System

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)

Source: http://www.csie.nctu.edu.tw/~wjtsai/EmbeddedSystemDesign/Ch3-1.pdf

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 */

inter.c

```
void __attribute__((interrupt)) irq_handler() {
 /* echo the received character + 1 */
 UART0_DR = UART0_DR + 1;
}
```

vectors.S

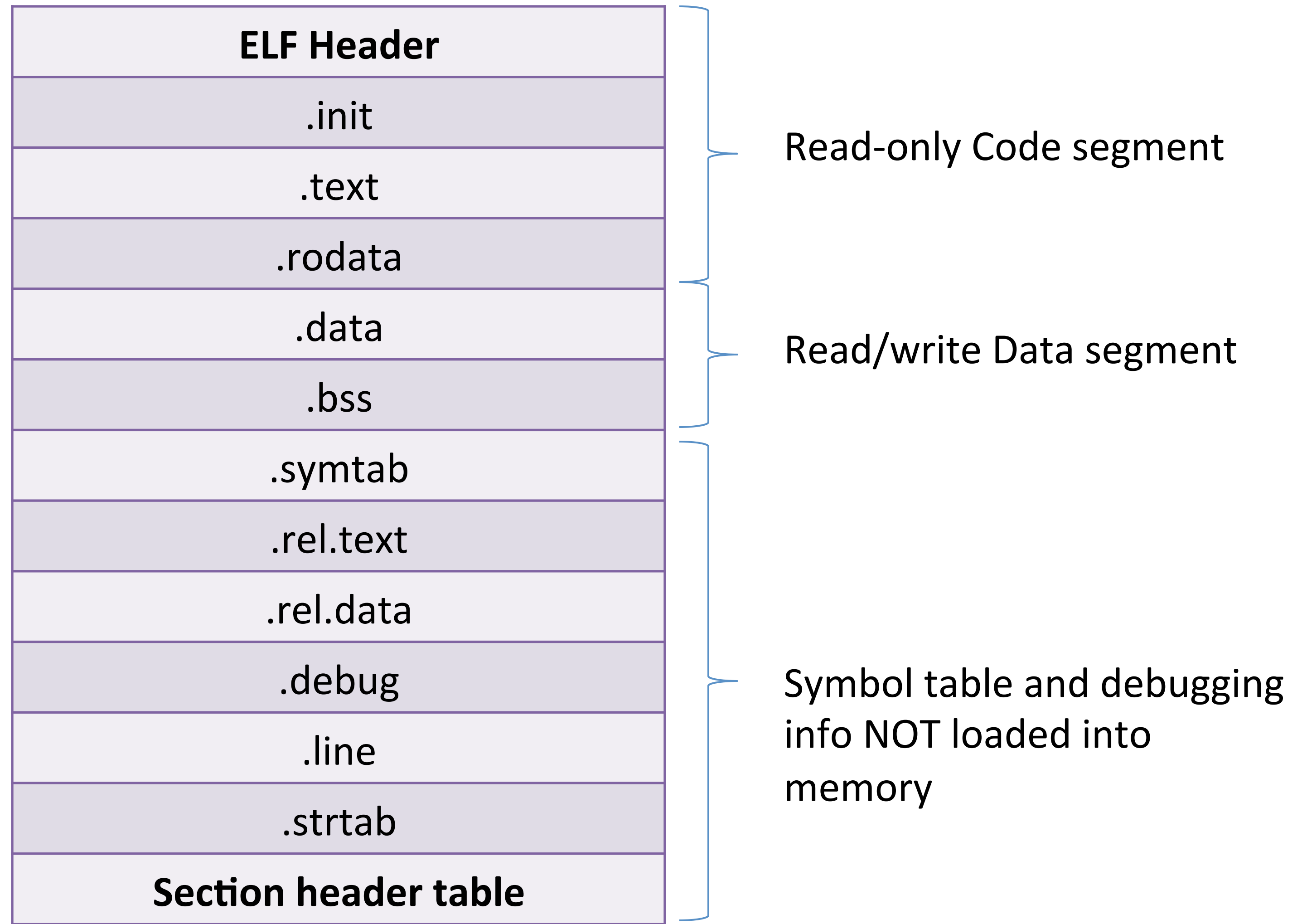
```
reset_handler:
 /* set Supervisor stack */
 LDR sp, =stack_top
 /* copy vector table to address 0 */
 BL copy_vectors
 /* get Program Status Register */
 MRS r0, cpsr
 /* go in IRQ mode */
 BIC r1, r0, #0x1F
 ORR r1, r1, #0x12
 MSR cpsr, r1
 /* set IRQ stack */
 LDR sp, =irq_stack_top
 /* Enable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

_Mode [4:0]	Mode
10000	User
10001	FIQ
10010	IRQ
10011	SVC (Supervisor)
10111	Abort
11011	Undefined
11111	System

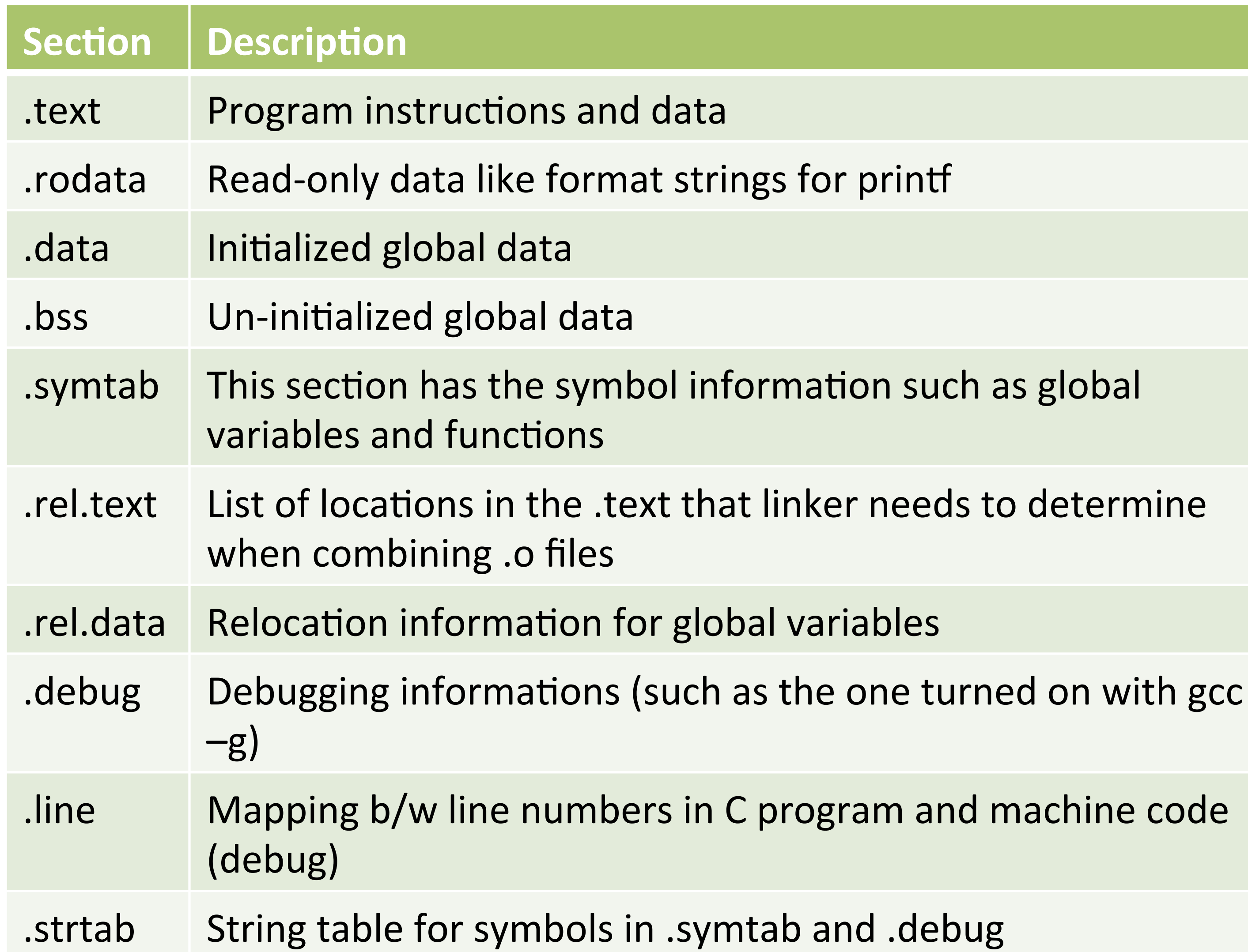
ARM ELF Format



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 grows towards increasing memory
- Stack starts at the opposite end and grows toward heap

ARM ELF Format



Section	Description
.text	Program instructions and data
.rodata	Read-only data like format strings for printf
.data	Initialized global data
.bss	Un-initialized global data
.symtab	This section has the symbol information such as global variables and functions
.rel.text	List of locations in the .text that linker needs to determine when combining .o files
.rel.data	Relocation information for global variables
.debug	Debugging informations (such as the one turned on with gcc –g)
.line	Mapping b/w line numbers in C program and machine code (debug)
.strtab	String table for symbols in .symtab and .debug

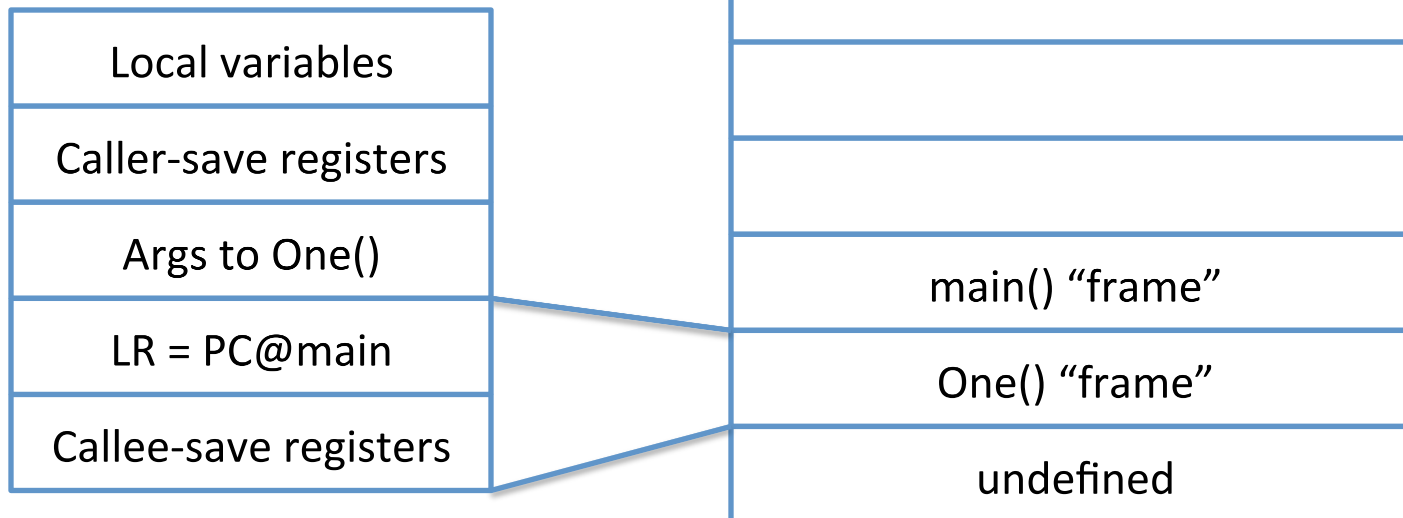
How to perform a control hijack

- 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



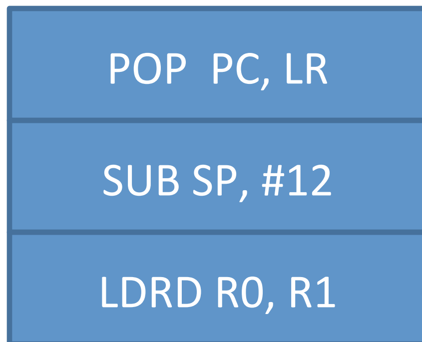
Itzhak Avraham's approach

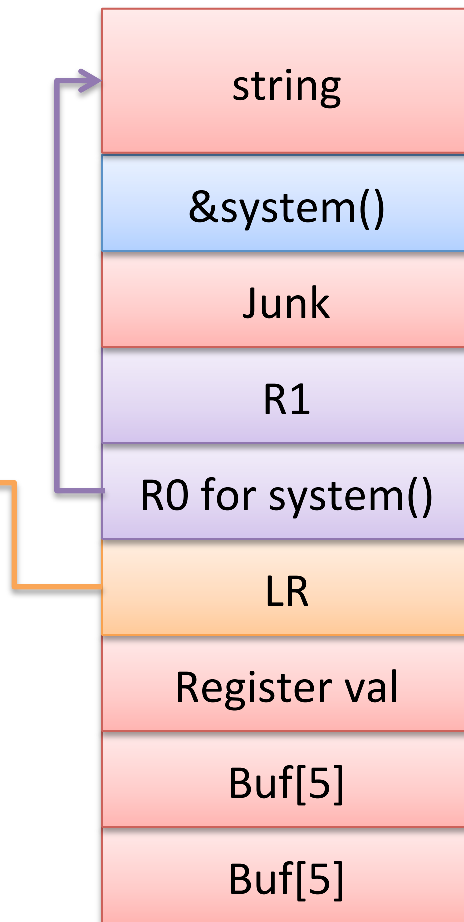
- Use a return to libc style method
- We can overwrite LR in stack
- Return to a function that contains instructions to pop values from stack into R0 (containing our "/bin/sh" address) and another LR in stack pointing to system()
- The above function that contains this code for us is erand48()

Source: https://media.blackhat.com/bh-dc-11/Avraham/BlackHat_DC_2011_Avraham-Popping_Android_Devices-Slides.pdf

Stack

erand48()+x:





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)

Increasing Memory

Lab 4

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

- Other useful links:
  - http://research.shell-storm.org/files/research-4-en.php

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”

DAY 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 NOT what system designer has in mind.

```
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 http://www.steike.com/code/useless/evil-c/

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

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

Source: http://gcc.gnu.org/onlinedocs/

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

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

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

  return sum;
}
```

```
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
        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

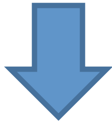
```
void t3(void)
{
        char c;
        int x=0;
        for(c=0;c<63;c++)
                x++;
}
```

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

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

ARM specific optimization: function calls

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



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

The following case shows square() has a side effect:

```
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

__pure int square(int x);

ARM specific optimization: code alignment

- Structure/Code alignment

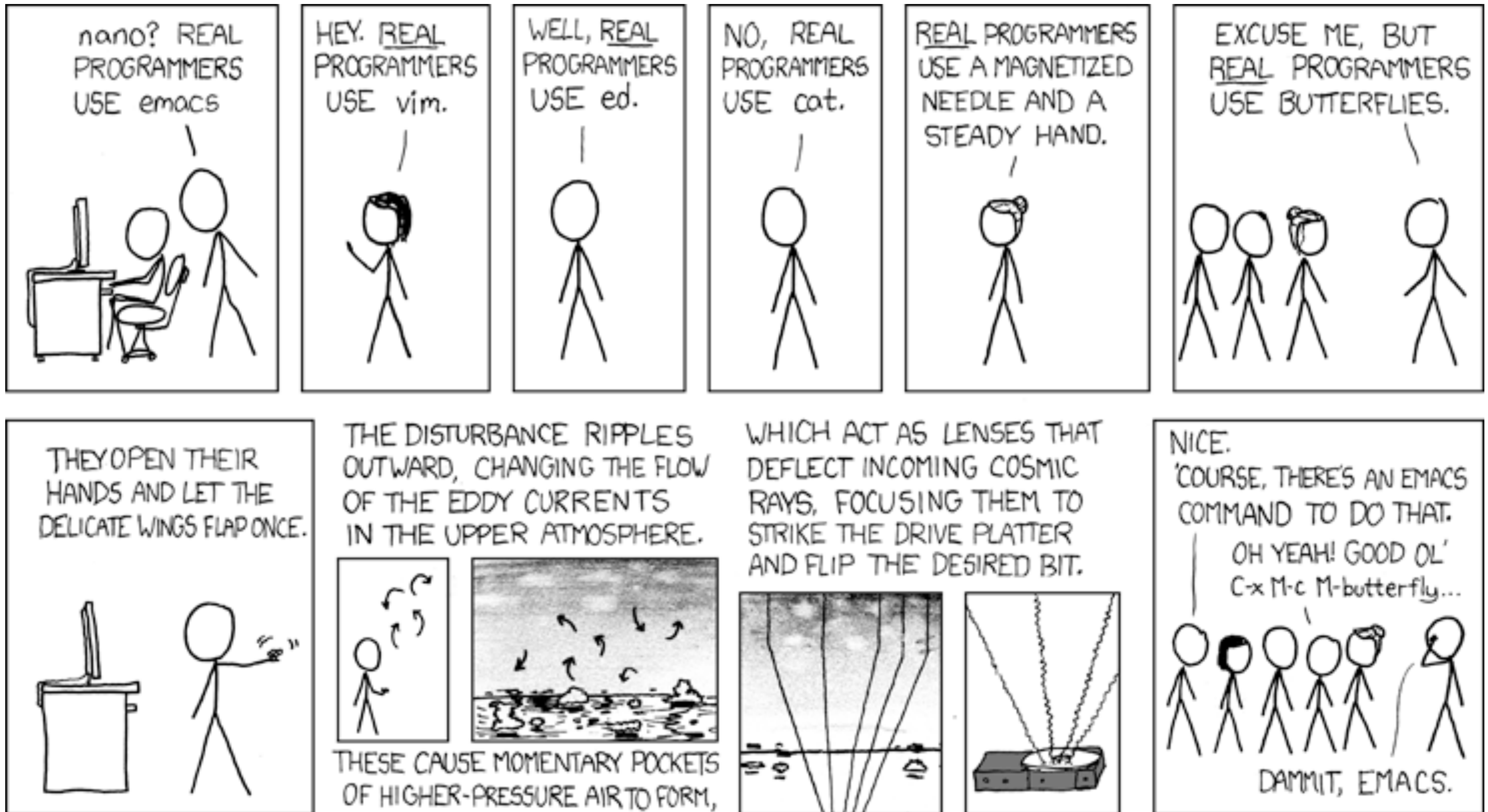
```
struct
{
        char a;
        int b;
        char c;
        short d;
}
```

```
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…



Source: http://xkcd.com/378/

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

Source: http://www.ethernut.de/en/documents/arm-inline-asm.html

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

Source: http://www.ethernut.de/en/documents/arm-inline-asm.html

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

Source: http://www.ethernut.de/en/documents/arm-inline-asm.html

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

```
0000838c <main>:
   838c:   b590        push    {r4, r7, lr}
   838e:   b085        sub     sp, #20
   8390:   af00        add     r7, sp, #0
   8392:   f04f 0306   mov.w   r3, #6
   8396:   60fb        str     r3, [r7, #12]
   8398:   f3ef 8400   mrs     r4, CPSR
   839c:   60bc        str     r4, [r7, #8]
   839e:   68fa        ldr     r2, [r7, #12]
   83a0:   f243 535d   movw    r3, #13661  ; 0x355d
   83a4:   f6cf 73fd   movt    r3, #65533  ; 0xfffd
   83a8:   18d3        adds    r3, r2, r3
   83aa:   607b        str     r3, [r7, #4]
   83ac:   f3ef 8400   mrs     r4, CPSR
   83b0:   603c        str     r4, [r7, #0]
   83b2:   f248 4344   movw    r3, #33860  ; 0x8444
   83b6:   f2c0 0300   movt    r3, #0
   83ba:   4618        mov     r0, r3
   83bc:   6879        ldr     r1, [r7, #4]
   ...
```

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

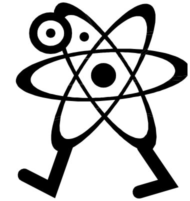
Exclusive instructions

- 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

        @.align 2
        @.thumb
        @.thumb_func
```

```
        .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:
    - http://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/iPhoneOSABIReference.pdf

Source:

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:
http://www.eggwall.com/2011/09/android-arm-assembly-calling-assembly.html

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
  - http://www.ethernut.de/en/documents/arm-inline-asm.html
- Writing ARM assembly
  - http://omappedia.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:
  - https://developer.mozilla.org/en/Developer_Guide/Virtual_ARM_Linux_environment
- GCC manual (ARM optimization options):
  - http://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html