Lecture 3: Assembly, Tools, and ABI
January 15, 2015
Announcements

- Homework 2 was posted on 1/13 is due on 1/20

- No office hours next week
Today...

Walk though of the ARM ISA

Software Development Tool Flow

Application Binary Interface (ABI)
The ARM architecture “books” for this class

- ARMv7-M Architecture Reference Manual
- Actel SmartFusion™ Microcontroller Subsystem User’s Guide
- SmartFusion Evaluation Kit User’s Guide
- Actel SmartFusion™ Programmable Analog User’s Guide
The ARM software tools “books” for this class

Sourcery G++ Lite
ARM EABI
Sourcery G++ Lite 2010q1-188
Getting Started
Exercise:
What is the value of r2 at done?

...  
start:  
    movs r0, #1  
    movs r1, #1  
    movs r2, #1  
    sub  r0, r1  
    bne  done  
    movs r2, #2  

done:  
    b    done  

...
Updating the APSR

- **SUB Rx, Ry**
  - Rx = Rx - Ry
  - APSR unchanged

- **SUBS**
  - Rx = Rx - Ry
  - APSR N, Z, C, V updated

- **ADD Rx, Ry**
  - Rx = Rx + Ry
  - APSR unchanged

- **ADDS**
  - Rx = Rx + Ry
  - APSR N, Z, C, V updated
Application Program Status Register (APSR)

APSR bit fields are in the following two categories:

- Reserved bits are allocated to system features or are available for future expansion. Further information on currently allocated reserved bits is available in *The special-purpose program status registers (xPSR)* on page B1-8. Application level software must ignore values read from reserved bits, and preserve their value on a write. The bits are defined as UNK/SBZP.

- Flags that can be set by many instructions:
  
  **N**, bit [31] Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then \( N = 1 \) if the result is negative and \( N = 0 \) if it is positive or zero.

  **Z**, bit [30] Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

  **C**, bit [29] Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.

  **V**, bit [28] Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

  **Q**, bit [27] Set to 1 if an SSAT or USAT instruction changes (saturates) the input value for the signed or unsigned range of the result.
## Conditional execution:
Append to many instructions for conditional execution

### Table A6-1 Condition codes

<table>
<thead>
<tr>
<th>cond</th>
<th>Mnemonic extension</th>
<th>Meaning (integer)</th>
<th>Meaning (floating-point) ab</th>
<th>Condition flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>Equal</td>
<td>Equal</td>
<td>Z == 1</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>Not equal</td>
<td>Not equal, or unordered</td>
<td>Z == 0</td>
</tr>
<tr>
<td>0010</td>
<td>CS c</td>
<td>Carry set</td>
<td>Greater than, equal, or unordered</td>
<td>C == 1</td>
</tr>
<tr>
<td>0011</td>
<td>CC d</td>
<td>Carry clear</td>
<td>Less than</td>
<td>C == 0</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td>Minus, negative</td>
<td>Less than</td>
<td>N == 1</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>Plus, positive or zero</td>
<td>Greater than, equal, or unordered</td>
<td>N == 0</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td>Overflow</td>
<td>Unordered</td>
<td>V == 1</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td>No overflow</td>
<td>Not unordered</td>
<td>V == 0</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td>Unsigned higher</td>
<td>Greater than, or unordered</td>
<td>C == 1 and Z == 0</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>Unsigned lower or same</td>
<td>Less than or equal</td>
<td>C == 0 or Z == 1</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td>Signed greater than or equal</td>
<td>Greater than or equal</td>
<td>N == V</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>Signed less than</td>
<td>Less than, or unordered</td>
<td>N != V</td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td>Signed greater than</td>
<td>Greater than</td>
<td>Z == 0 and N == V</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td>Signed less than or equal</td>
<td>Less than, equal, or unordered</td>
<td>Z == 1 or N != V</td>
</tr>
<tr>
<td>1110</td>
<td>None (AL) e</td>
<td>Always (unconditional)</td>
<td>Always (unconditional)</td>
<td>Any</td>
</tr>
</tbody>
</table>
Solution:
what is the value of r2 at **done**?

...  

**start:**

```assembly
movs r0, #1 // r0 ← 1, Z=0
movs r1, #1 // r1 ← 1, Z=0
movs r2, #1 // r2 ← 1, Z=0
sub r0, r1 // r0 ← r0-r1
    // **but Z flag untouched**
    // **since sub vs sub**
    // **NE true when Z==0**
    // **So, take the branch**
bne done   // NE true when Z==0
    // **not executed**
```

**done:**

```assembly
movs r2, #2 // r2 is still 1
```

...
Real assembly example

.equ STACK_TOP, 0x20000000
.text
.syntax unified
.thumb
.global _start

.type start, %function

_start:
   .word STACK_TOP, start

start:
   movs r0, #10
   movs r1, #0

loop:
   adds r1, r0
   subs r0, #1
   bne loop

deadloop:
   b  deadloop
.end
What’s it all mean?

```assembly
.equ STACK_TOP, 0x20000800  /* Sets symbol to value (#define)*/
.text  /* Tells AS to assemble region */
.syntax unified  /* Means language is ARM UAL */
.thumb  /* Means ARM ISA is Thumb */
.global _start  /* .global exposes symbol */
    /* _start label is the beginning */
    /* ...of the program region */
    /* Specifies start is a function */
    /* start label is reset handler */
.type start, %function  /* Specifies start is a function */

_start:
.word STACK_TOP, start  /* Inserts word 0x20000800 */

start:
    movs r0, #10  /* We’ve seen the rest ... */
    movs r1, #0

loop:
    adds r1, r0
    subs r0, #1
    bne loop

deadloop:
    b deadloop
.end
```
What happens after a power-on-reset (POR)?

- **ARM Cortex-M3** (many others are similar)

- **Reset procedure**
  - SP ← mem(0x00000000)
  - PC ← mem(0x00000004)

```
_start:
    .word __STACKTOP          /* Top of Stack */
    .word Reset_Handler       /* Reset Handler */
    .word NMI_Handler         /* NMI Handler */
    .word HardFault_Handler   /* Hard Fault Handler */
    .word MemManage_Handler   /* MPU Fault Handler */
    .word BusFault_handler    /* Bus Fault Handler */
    ...
```
Today...

Walk though of the ARM ISA

Software Development Tool Flow

Application Binary Interface (ABI)
How does an assembly language program get turned into a executable program image?

Assembly files (.s) → Object files (.o) → Linker (ld) → Executable image file

- Binary program file (.bin)
- Disassembled code (.lst)
- Memory layout
- Linker script (.ld)
- Objcopy
- Objdump
What are the real GNU executable names for the ARM?

- Just add the prefix “arm-none-eabi-” prefix
- Assembler (as)
  - arm-none-eabi-as
- Linker (ld)
  - arm-none-eabi-ld
- Object copy (objcopy)
  - arm-none-eabi-objcopy
- Object dump (objdump)
  - arm-none-eabi-objdump
- C Compiler (gcc)
  - arm-none-eabi-gcc
- C++ Compiler (g++)
  - arm-none-eabi-g++
Real-world example

• To the terminal!

(code at https://github.com/brghena/eecs373_toolchain_examples)
How are assembly files assembled?

- $ arm-none-eabi-as
  - Useful options
    - -mcpu
    - -mthumb
    - -o

$ arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
A simple (hardcoded) Makefile example

```make
all:
    arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
    arm-none-eabi-ld -Ttext 0x0 -o example1.out example1.o
    arm-none-eabi-objcopy -Obinary example1.out example1.bin
    arm-none-eabi-objdump -S example1.out > example1.lst
```
What information does the disassembled file provide?

```
all:
    arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
    arm-none-eabi-ld -Ttext 0x0 -o example1.out example1.o
    arm-none-eabi-objcopy -Obinary example1.out example1.bin
    arm-none-eabi-objdump -S example1.out > example1.lst
```

```
.equ STACK_TOP, 0x20000800
.text
.syntax unified
.thumb
.global _start
.type start, %function

_start:
    .word STACK_TOP, start

start:
    movs r0, #10
    movs r1, #0

loop:
    adds r1, r0
    subs r0, #1
    bne loop

deadloop:
    b deadloop
.end

example1.out:     file format elf32-littlearm

Disassembly of section .text:

```
00000000 <_start>:
  0:  20000800 .word 0x20000800
  4:  00000009 .word 0x00000009

00000008 <start>:
  8:  200a movs r0, #10
  a:  2100 movs r1, #0

000000c <loop>:
  c:  1809 adds r1, r1, r0
  e:  3801 subs r0, #1
  10: d1fc bne.n c <loop>

0000012 <deadloop>:
  12: e7fe b.n 12 <deadloop>
```
Linker script

OUTPUT_FORMAT("elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(main)

MEMORY
{
    /* SmartFusion internal eSRAM */
    ram (rwx) : ORIGIN = 0x20000000, LENGTH = 64k
}

SECTIONS
{
    .text :
    {
        . = ALIGN(4);
        *(.text*)
        . = ALIGN(4);
        _etext = .;
    } >ram
}

end = .;

• Specifies little-endian arm in ELF format.
• Specifies ARM CPU
• Should start executing at label named “main”
• We have 64k of memory starting at 0x20000000. You can read, write and execute out of it. We’ve named it “ram”

• “.” is a reference to the current memory location
• First align to a word (4 byte) boundary
• Place all sections that include .text at the start (* here is a wildcard)
• Define a label named _etext to be the current address.
• Put it all in the memory location defined by the ram memory location.
How does a mixed C/Assembly program get turned into an executable program image?

C files (.c) → as (assembler) → Object files (.o) → gcc (compile + link) → Executable image file

Assembly files (.s) → Library object files (.o) → Linker script (.ld) → ld (linker) → Binary program file (.bin)

Disassembled code (.lst) → objdump

objcopy
Real-world example #2

- To the terminal! Again!

(code at https://github.com/brghena/eecs373_toolchain_examples)
Today...

Finish ARM assembly example from last time

Walk though of the ARM ISA

Software Development Tool Flow

Application Binary Interface (ABI)
<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.</td>
</tr>
<tr>
<td>r14</td>
<td>LR</td>
<td></td>
<td>The Link Register.</td>
</tr>
<tr>
<td>r13</td>
<td>SP</td>
<td></td>
<td>The Stack Pointer.</td>
</tr>
<tr>
<td>r12</td>
<td>IP</td>
<td></td>
<td>The Intra-Procedure-call scratch register.</td>
</tr>
<tr>
<td>r11</td>
<td>v8</td>
<td></td>
<td>Variable-register 8.</td>
</tr>
<tr>
<td>r10</td>
<td>v7</td>
<td></td>
<td>Variable-register 7.</td>
</tr>
<tr>
<td>r9</td>
<td>v6</td>
<td>SB</td>
<td>Platform register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TR</td>
<td>The meaning of this register is defined by the platform standard.</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.</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>
ABI Basic Rules

1. A subroutine must preserve the contents of the registers r4-11 and SP
   - Let’s be careful with r9 though.

2. Arguments are passed though r0 to r3
   - If we need more, we put a pointer into memory in one of the registers.
     • We’ll worry about that later.

3. Return value is placed in r0
   - r0 and r1 if 64-bits.

4. Allocate space on stack as needed. Use it as needed. Put it back when done...
   - Keep word aligned.
When is this relevant?

• The ABI is a contract with the compiler
  - All assembled C code will follow this standard

• You need to follow it if you want C and Assembly to work together correctly

• What if you are writing everything in Assembly by hand?
  - Maybe less important. Unless you’re ever going to extend the code
Let’s write a simple ABI routine

- int bob(int a, int b)
  - returns $a^2 + b^2$

- Instructions you might need
  - add adds two values
  - mul multiplies two values
  - bx branch to register

Other useful facts
- Stack grows down.
  - And pointed to by “sp”

- Address we need to go back to in “lr”

<table>
<thead>
<tr>
<th>Register</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>r15</td>
<td></td>
</tr>
<tr>
<td>r14</td>
<td></td>
</tr>
<tr>
<td>r13</td>
<td></td>
</tr>
<tr>
<td>r12</td>
<td></td>
</tr>
<tr>
<td>r11</td>
<td>v8</td>
</tr>
<tr>
<td>r10</td>
<td>v7</td>
</tr>
<tr>
<td>r9</td>
<td></td>
</tr>
<tr>
<td>r8</td>
<td>v5</td>
</tr>
<tr>
<td>r7</td>
<td>v4</td>
</tr>
<tr>
<td>r6</td>
<td>v3</td>
</tr>
<tr>
<td>r5</td>
<td>v2</td>
</tr>
<tr>
<td>r4</td>
<td>v1</td>
</tr>
<tr>
<td>r3</td>
<td>a4</td>
</tr>
<tr>
<td>r2</td>
<td>a3</td>
</tr>
<tr>
<td>r1</td>
<td>a2</td>
</tr>
<tr>
<td>r0</td>
<td>a1</td>
</tr>
</tbody>
</table>
Questions?

Comments?

Discussion?