Today...

Finish ARM assembly example from last time
Walk though of the ARM ISA
Software Development Tool Flow
Application Binary Interface (ABI)

The endianess religious war: 288 years and counting!

- Modern version
  - Danny Cohen
  - IEEE Computer, v14, #10
  - Published in 1981
  - Satire on CS religious war

- Historical Inspiration
  - Jonathan Swift
  - Gulliver's Travels
  - Published in 1726
  - Satire on Henry-VIII's split with the Church
  - Now a major motion picture!

Addressing: Big Endian vs Little Endian (370 slide)

- Endian-ness: ordering of bytes within a word
  - Little - increasing numeric significance with increasing memory addresses
  - Big - The opposite, most significant byte first
  - MIPS is big endian, x86 is little endian
Instruction encoding

- Instructions are encoded in machine language opcodes
- Sometimes
  - Necessary to hand generate opcodes
  - Necessary to verify assembled code is correct
- How? Refer to the “ARM ARM”

```
<table>
<thead>
<tr>
<th>Instructions</th>
<th>Register Value</th>
<th>Memory Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>movs r0, #10</td>
<td>001</td>
<td>00</td>
</tr>
<tr>
<td>movs r1, #0</td>
<td>001</td>
<td>00</td>
</tr>
</tbody>
</table>
```

Assembly example

```
data:
.byte 0x12, 20, 0x20, -1

func:
  mov r0, #0
  mov r4, #0
  movw r1, #:lower16:data
  movt r1, #:upper16:data
  top:    ldrb r2, [r1],#1
          add r4, r4, r2
          add r0, r0, #1
          cmp r0, #4
          bne top
```
Loads!

- **ldrb** -- *Load register byte*
  - Note this takes an 8-bit value and moves it into a 32-bit location!
  - Zeros out the top 24 bits

- **ldrsb** -- *Load register signed byte*
  - Note this also takes an 8-bit value and moves it into a 32-bit location!
  - Uses sign extension for the top 24 bits

Addressing Modes

- **Offset Addressing**
  - Offset is added or subtracted from base register
  - Result used as effective address for memory access
  - `[<Rn>, <offset>]`

- **Pre-indexed Addressing**
  - Offset is applied to base register
  - Result used as effective address for memory access
  - Result written back into base register
  - `[<Rn>, <offset>]`

- **Post-indexed Addressing**
  - The address from the base register is used as the EA
  - The offset is applied to the base and then written back
  - `[<Rn>], <offset>`

---

So what does the program _do_?

data:
  .byte 0x12, 20, 0x20, -1
func:
  mov r0, #0
  mov r4, #0
  movw r1, #:lower16:data
  movt r1, #:upper16:data
  top:    ldrb r2, [r1],#1
  add r4, r4, r2
  add r0, r0, #1
  cmp r0, #4
  bne top

Today...

Finish ARM assembly example from last time

Walk through of the ARM ISA

Application Binary Interface (ABI)

An ISA defines the hardware/software interface

- A “contract” between architects and programmers
- Register set
- Instruction set
  - Addressing modes
  - Word size
  - Data formats
  - Operating modes
  - Condition codes
- **Calling conventions**
  - Really not part of the ISA (usually)
  - Rather part of the ABI
  - But the ISA often provides meaningful support.

ARM Architecture roadmap
A quick comment on the ISA:

4.1 About the instruction set
ARMv7-M supports a large number of 32-bit instructions that were introduced in Thumb-2 technology into the Thumb instruction set. Most of the functionality available is identical to the ARM instruction set. See Chapter 8 for an introduction to Thumb-2 and Chapter 9 for a detailed description of Thumb-2 instructions. This chapter describes the functionality available in the ARMv7-M Thumb instruction set and the Thumb processor.

Thum instruction set includes:
- 16-bit instructions
- 32-bit instructions
- 64-bit instructions

Most Thumb instructions are 16-bit instructions that can be executed in Thumb mode.

Branching
Data processing
Load/Store
Exceptions
Miscellaneous

Instruction Set
Register Set
Address Space

Address Space:
- System
- On-chip memory
- External memory
- General purpose registers
- Link register

Instruction Encoding
ADD immediate

Registers
SP_main (MSP) used by:
- OS kernel
- Exception handlers
- App code w/ privileged access

SP_process (PSP) used by:
- Base app code (when not running an exception handler)

Note: there are two stack pointers!

Mode dependent

20
Branch

Table A4-1 Branch instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Usage</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>B on page A4-40</td>
<td>Branch to target address</td>
<td>+/−1 B</td>
</tr>
<tr>
<td>CBNZ, CBE on page A6-52</td>
<td>Compare and Branch on Zero, Compare and Branch on Zero</td>
<td>0-125 B</td>
</tr>
<tr>
<td>BL on page A4-49</td>
<td>Call a subroutine</td>
<td>+/−16 MB</td>
</tr>
<tr>
<td>BLC (register) on page A6-50</td>
<td>Call a subroutine, optionally change instruction set</td>
<td>Any</td>
</tr>
<tr>
<td>BX on page A6-51</td>
<td>Branch to target address, change instruction set</td>
<td>Any</td>
</tr>
<tr>
<td>TBRR, TBHF on page A6-358</td>
<td>Table Branch (byte-offset)</td>
<td>0-125 B</td>
</tr>
<tr>
<td>Table Branch (halfword-offset)</td>
<td></td>
<td>0-1257 MB</td>
</tr>
</tbody>
</table>

Many, Many More!

Load/Store instructions

Table A4-10 Load and store instructions

<table>
<thead>
<tr>
<th>Data type</th>
<th>Load privile/ Store privile/ Load exclusive/ Store exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit word</td>
<td>LDH                  STH                  LOHT                 STHX</td>
</tr>
<tr>
<td>16-bit halfword</td>
<td>STH                  HTH                  HORT                 STHX</td>
</tr>
<tr>
<td>16-bit unaligned halfword</td>
<td>LDH                  STH                  LOHT                 STHX</td>
</tr>
<tr>
<td>16-bit signed halfword</td>
<td>LDH                  STH                  LOHT                 STHX</td>
</tr>
<tr>
<td>8-bit byte</td>
<td>LDR                  STR                  LDRF                 STRX</td>
</tr>
<tr>
<td>8-bit unaligned byte</td>
<td>LDR                  STR                  LDRF                 STRX</td>
</tr>
<tr>
<td>8-bit signed byte</td>
<td>LDR                  STR                  LDRF                 STRX</td>
</tr>
<tr>
<td>two 32-bit words</td>
<td>LDW                  SWD                  LOHDX                STHX</td>
</tr>
</tbody>
</table>

Miscellaneous instructions

Table A4-12 Miscellaneous instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLZ</td>
<td>CLZ on page A6-58</td>
</tr>
<tr>
<td>Branch</td>
<td>BRK on page A6-65</td>
</tr>
<tr>
<td>Branch</td>
<td>BSR on page A6-68</td>
</tr>
<tr>
<td>Branch</td>
<td>BSR on page A6-70</td>
</tr>
<tr>
<td>Instruction Type</td>
<td>BRK on page A6-70</td>
</tr>
<tr>
<td>Instruction Type</td>
<td>BRK on page A6-70</td>
</tr>
<tr>
<td>Instruction Type</td>
<td>BRK on page A6-70</td>
</tr>
<tr>
<td>Instruction Type</td>
<td>BRK on page A6-70</td>
</tr>
<tr>
<td>Instruction Type</td>
<td>BRK on page A6-70</td>
</tr>
<tr>
<td>Instruction Type</td>
<td>BRK on page A6-70</td>
</tr>
<tr>
<td>Instruction Type</td>
<td>BRK on page A6-70</td>
</tr>
</tbody>
</table>

Addressing Modes (again)

- Offset Addressing
  - Offset is added or subtracted from base register
  - Result used as effective address for memory access
    - [<Rn>, <offset>]

- Pre-indexed Addressing
  - Offset is applied to base register
  - Result used as effective address for memory access
  - Result written back into base register
    - [<Rn>, <offset>]

- Post-indexed Addressing
  - The address from the base register is used as the EA
  - The offset is applied to the base and then written back
    - [<Rn>], <offset>

<offset> options

- An immediate constant
  - #10

- An index register
  - <Rm>

- A shifted index register
  - <Rm>, LSL #<shift>

- Lots of weird options...
**Application Program Status Register (APSR)**

APSR bit fields are in the following two categories:

- Reserved bits are allocated to express features or are available for future expansion. Further information on currently allocated reserved bits is available in the specific purpose status register in the ARM architecture reference manual on page 318. Application level software must ensure that these bit fields are preserved in the APSR, and provide their value as a return. The bit fields are defined in the ARM architecture reference manual.

- Flags that can be set by many instructions:
  - N, bit [31] Negative carry flag.
  - V, bit [31] Overflow flag.

Overflow and carry in APSR:

```plaintext
unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
result = unsigned_sum<N-1:0>; // == signed_sum<N-1:0>
carry_out = if UInt(result) == unsigned_sum then '0' else '1';
overflow = if SInt(result) == signed_sum then '0' else '1';
```

**Conditional execution:**

Append to many instructions for conditional execution:

<table>
<thead>
<tr>
<th>cond</th>
<th>Mnemonic extension</th>
<th>Meaning (integer)</th>
<th>Meaning (floating-point)</th>
<th>Condition flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>01</td>
<td>Equal</td>
<td>Z = 1</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>01</td>
<td>Not equal</td>
<td>Not equal, unordered</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>01</td>
<td>Carry</td>
<td>Greater than, equal, ordered</td>
<td>C = 1</td>
</tr>
<tr>
<td>0011</td>
<td>01</td>
<td>Less than</td>
<td>C = 0</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>01</td>
<td>Signed zero</td>
<td>Less than, equal, unordered</td>
<td>N = 1</td>
</tr>
<tr>
<td>0101</td>
<td>01</td>
<td>Signed positive</td>
<td>Greater than, equal, unordered</td>
<td>N = 0</td>
</tr>
<tr>
<td>0110</td>
<td>01</td>
<td>Overflow</td>
<td>Unordered</td>
<td>V = 1</td>
</tr>
<tr>
<td>1110</td>
<td>01</td>
<td>Signed negative</td>
<td>Not unordered</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11</td>
<td>Signed greater than</td>
<td>Greater than, equal</td>
<td>N = V</td>
</tr>
<tr>
<td>0101</td>
<td>11</td>
<td>Signed less than</td>
<td>Less than, equal</td>
<td>N = V</td>
</tr>
<tr>
<td>1100</td>
<td>11</td>
<td>Signed unordered</td>
<td>Not ordered</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>11</td>
<td>Signed zero</td>
<td>Z = 0 and N = 0</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>11</td>
<td>Signed negative</td>
<td>Greater than, equal, unordered</td>
<td>Z = 1 and N = V</td>
</tr>
<tr>
<td>1111</td>
<td>11</td>
<td>Signed greater than</td>
<td>Greater than, equal</td>
<td>N = V</td>
</tr>
<tr>
<td>0000</td>
<td>00</td>
<td>Always (unconditional)</td>
<td>Always (unconditional)</td>
<td>Any</td>
</tr>
</tbody>
</table>

**The ARM architecture “books” for this class**

- Online ARM Manual
- ARM Architecture Manual

**ARMv7-M Architecture Reference Manual**

ARMv7-M_ARM.pdf

**Application Program Status Register (APSR)**

- 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

- Overflow and carry in APSR

```plaintext
unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
result = unsigned_sum<N-1:0>; // == signed_sum<N-1:0>
carry_out = if UInt(result) == unsigned_sum then '0' else '1';
overflow = if SInt(result) == signed_sum then '0' else '1';
```
The ARM software tools “books” for this class

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
...!

Solution:
what is the value of r2 at done?

...!
start:
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
bne done // NE true when Z=0
// So, take the branch
movs r2, #2 // not executed
done:
b done // r2 is still 1
...

Today...

Finish ARM assembly example from last time
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?

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++
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 "real" ARM assembly language program for GNU

```c
.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
```

A simple (hardcoded) Makefile example

```c
all:
    arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
```

What's it all mean?

```
.equ STACK_TOP, 0x20000000 /* Equates symbol to value */
.text /* Tells AS to assemble region */
.syntax unified /* Means language is ARM UAL */
.thumb /* Means ARM ISA is Thumb */
.global _start /* .global exposes symbol */
.type start, %function /* start label is the beginning */
_start: /* ....of the program region */
.word STACK_TOP, start /* Specifies start is a function */
_start: /* start label is reset handler */
    movs r0, #10
    movs r1, #0
start: /* We've seen the rest ... */
    adds r1, r0
    subs r0, #1
    bne start
loop: /* */
    moves r0, #10
    moves r1, #0
    bne loop
deadloop: /* */
    b deadloop
.end
```

How does a mixed C/Assembly program get turned into an executable program image?

**Assembly files (.s)**

- Disassembled Code (.lst)
- Library object files (.o)
- Linker script (.ld)
- C files (.c)

**Object files (.o)**

- Linking

**Binary program file (.bin)**

- Executable image file

**Executable image file**

- Linking

**Libraries**

- Assembly files (.s)

**Assembly files (.s)**

- Disassembled Code (.lst)
- Library object files (.o)
- Linker script (.ld)
- C files (.c)
Today...

Finish ARM assembly example from last time
Walk though of the ARM ISA
Software Development Tool Flow
Application Binary Interface (ABI)

ABI quote

- A subroutine must preserve the contents of the registers r4-r8, r10, r11 and SP (and r9 in PCS variants that designate r9 as v6).

Questions?
Comments?
Discussion?