EECS 373
Design of Microprocessor-Based Systems

Prabal Dutta
University of Michigan

Lecture 2: Architecture, Assembly, and ABI
January 13, 2015

Slides developed in part by Mark Brehob

Announcements

• Website up
  - http://www.eecs.umich.edu/~prabal/teaching/eecs373/

• Homework 2 posted (mostly a 370 review)

• Lab and office hours posted on-line.
  - My office hours: Thursday 3:00-4:00 pm in 4773 BBB

• Projects
  - Start thinking about them now!

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!

Little-Endian
  - LSB is at lower address

Big-Endian
  - MSB is at lower address

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
ARM Cortex-M3 Memory Formats (Endian)

- Default memory format for ARM CPUs: LITTLE ENDIAN
- Bytes 0-3 hold the first stored word
- Bytes 4-7 hold the second stored word
- Processor contains a configuration pin BIGEND
  - Enables system developer to select format:
    - Little Endian
    - Big Endian (BE-8)
  - Pin is sampled on reset
  - Cannot change endianness when out of reset


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

```assembly
.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
```

Instructions used

- **mov**
  - Moves data from register or immediate.
  - Or also from shifted register or immediate!
    - the mov assembly instruction maps to a bunch of different encodings!
    - If immediate it might be a 16-bit or 32-bit instruction
    - Not all values possible
    - why?
- **movw**
  - Actually an alias to mov
    - “w” is “wide”
    - hints at 16-bit immediate

Directives

- #:lower16:data
  - What does that do?
  - Why?
- Note:
  - “data” is a label for a memory address!

There are similar entries for move immediate, move shifted (which actually maps to different instructions) etc.
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

- **ldrb** -- 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

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.
A quick comment on the ISA:

A.4.1 About the instruction set

ARMv7-M supports a large number of 32-bit instructions that were introduced as Thumb-2 technology into the Thumb instruction set. Much of the functionality available is similar to the ARM instruction set, but an expanded instruction set is supported alongside the Thumb instruction set in ARMv7-M and other ARM7 profiles. This chapter describes the functionality available in the ARMv7-M Thumb instruction set, and the Unified Assembly Language (UAL) that can be assembled to either the Thumb or ARM instruction sets.

Thumb instructions are either 16-bit or 32-bit, and are aligned on a two-byte boundary. 16-bit and 32-bit instructions can be terminated freely. Many common operations are more efficiently executed using 16-bit instructions. However:

- Most 16-bit instructions can only access eight of the general purpose registers, R0-R7. These are known as the low registers. A small number of 16-bit instructions can access the high registers, R8-R13.
- Many operations that would require two or more 16-bit instructions can be more efficiently executed with a single 32-bit instruction.

The ARM and Thumb instruction sets are designed to work together. Because ARMv7-M only supports Thumb instructions, interrupting instructions in ARMv7-M must only reference Thumb state execution, see ARMv7-M and interrupting support for more details.

In addition, see:
- Chapter 15 Thumb Instruction Set Encoding for encoding details of the Thumb instruction set.
- Chapter 16 Thumb Instruction Details for detailed descriptions of the instructions.

ARM Cortex-M3 ISA

Instruction Set
ADD rd, rn, <op2>

Branching
Data processing
Load/Store
Exceptions
Miscellaneous

Register Set
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12

R13 (SP)
R14 (LR)
R15 (PC)

SPSR

Address Space
32-bits
Endianness

32-bits
Endianness

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

SP, main (MSP) used by:
- OS kernel
- Exception handlers
- App code w/ privileged access

Note: there are two stack pointers!

Mode dependent

Instruction Encoding
ADD immediate

Encoding T1
All versions of the Thumb ISA

<table>
<thead>
<tr>
<th>Imm</th>
<th>Rdimi</th>
<th>RSimi</th>
<th>OP0imi</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding T2
All versions of the Thumb ISA

<table>
<thead>
<tr>
<th>Imm</th>
<th>Rdimi</th>
<th>RSimi</th>
<th>OP0imi</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding T3
ARMv7-M

<table>
<thead>
<tr>
<th>Imm</th>
<th>Rdimi</th>
<th>RSimi</th>
<th>OP0imi</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding T4
ARMv7-M

<table>
<thead>
<tr>
<th>Imm</th>
<th>Rdimi</th>
<th>RSimi</th>
<th>OP0imi</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding T5
ARMv7-M

<table>
<thead>
<tr>
<th>Imm</th>
<th>Rdimi</th>
<th>RSimi</th>
<th>OP0imi</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding T6
ARMv7-M

<table>
<thead>
<tr>
<th>Imm</th>
<th>Rdimi</th>
<th>RSimi</th>
<th>OP0imi</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Branch examples

- **b target**
  - Branch without link (i.e. no possibility of return) to target
  - The PC is not saved!
- **bl func**
  - Branch with link (call) to function `func`
  - Store the return address in the link register (`lr`)
- **bx lr**
  - Use to return from a function
  - Moves the `lr` value into the `pc`
  - Could be a different register than `lr` as well
- **blx reg**
  - Branch to address specified by `reg`
  - Save return address in `lr`
  - When using `blx`, make sure lsb of `reg` is 1 (otherwise, the CPU will fault b/c it’s an attempt to go into the ARM state)

Branch examples (2)

- **blx label**
  - Branch with link and exchange state. With immediate data, `blx` changes to ARM state. But since CM-3 does not support ARM state, this instruction causes a fault!
- **mov r15, r0**
  - Branch to the address contained in `r0`
- **ldr r15, [r0]**
  - Branch to the address in memory specified by `r0`

Calling `bl` overwrites contents of `lr`!
- So, save `lr` if your function needs to call a function!

Data processing instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td><code>ADD</code></td>
<td>Two 32-bit operands are added and result is stored in the destination register.</td>
</tr>
<tr>
<td>SUB</td>
<td><code>SUB</code></td>
<td>Two 32-bit operands are subtracted and result is stored in the destination register.</td>
</tr>
<tr>
<td>LDR</td>
<td><code>LDR</code></td>
<td>Loads a 32-bit value from memory into the destination register.</td>
</tr>
<tr>
<td>STR</td>
<td><code>STR</code></td>
<td>Stores the value in the source register into memory at the specified address.</td>
</tr>
</tbody>
</table>

Load/Store instructions

<table>
<thead>
<tr>
<th>Data type</th>
<th>Load</th>
<th>Store</th>
<th>Load unprivileged</th>
<th>Store unprivileged</th>
<th>Load exclusive</th>
<th>Store exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit word</td>
<td>LDR</td>
<td>STR</td>
<td>LDR/STR</td>
<td>LDR/STR</td>
<td>LDR/STR/STREX</td>
<td>LDR/STR/STREX</td>
</tr>
<tr>
<td>16-bit halfword</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-bit unsigned halfword</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-bit signed halfword</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-bit byte</td>
<td>STR</td>
<td></td>
<td>LDR/STR</td>
<td>LDR/STR</td>
<td>LDR/STR/STREX</td>
<td>LDR/STR/STREX</td>
</tr>
<tr>
<td>8-bit unsigned byte</td>
<td>LDR</td>
<td></td>
<td>LDR/STR</td>
<td>LDR/STR</td>
<td>LDR/STR/STREX</td>
<td>LDR/STR/STREX</td>
</tr>
<tr>
<td>8-bit signed byte</td>
<td>LDR</td>
<td></td>
<td>LDR/STR</td>
<td>LDR/STR</td>
<td>LDR/STR/STREX</td>
<td>LDR/STR/STREX</td>
</tr>
<tr>
<td>16-bit word</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit signed word</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit words</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Many, Many More!
### Miscellaneous instructions

#### Table A1-12: Miscellaneous instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear Flags</td>
<td>CLRZ on page A1-16</td>
</tr>
<tr>
<td>Debug Exit</td>
<td>DSB on page A8-67</td>
</tr>
<tr>
<td>Data Memory Barrier</td>
<td>DMB on page A6-60</td>
</tr>
<tr>
<td>Data Memory Barrier</td>
<td>DMB on page A7-60</td>
</tr>
<tr>
<td>Instruction, Synchronization Barrier</td>
<td>ISB on page A6-70</td>
</tr>
<tr>
<td>IF True (before executing instructions conditioned)</td>
<td>IT on page A6-70</td>
</tr>
<tr>
<td>No Operation</td>
<td>NOP on page A6-167</td>
</tr>
<tr>
<td>Parity Error</td>
<td>PE on page A6-176</td>
</tr>
<tr>
<td>Parity Instruction</td>
<td>PLR on page A6-182</td>
</tr>
<tr>
<td>PL instruction</td>
<td>PLo on page A6-186</td>
</tr>
<tr>
<td>Read Error</td>
<td>SET on page A8-222</td>
</tr>
<tr>
<td>Instruction Fail</td>
<td>JFL on page A8-222</td>
</tr>
<tr>
<td>Set for Interrupt</td>
<td>SFI on page A8-276</td>
</tr>
<tr>
<td>Yield</td>
<td>YIELD on page A8-278</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)

#### 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

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>Meaning (integer)</th>
<th>Meaning (floating-point)</th>
<th>Condition flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Eq</td>
<td>Equal</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>Not equal, unordered</td>
<td>Z = 0</td>
</tr>
<tr>
<td>0010</td>
<td>GT</td>
<td>Greater than, equal, or unordered</td>
<td>Z = 1</td>
</tr>
<tr>
<td>0011</td>
<td>GE</td>
<td>Greater than or equal</td>
<td>Z = C = 0</td>
</tr>
<tr>
<td>0100</td>
<td>LE</td>
<td>Less than, equal</td>
<td>N = 1</td>
</tr>
<tr>
<td>0101</td>
<td>LT</td>
<td>Less than, unordered</td>
<td>N = V</td>
</tr>
<tr>
<td>0110</td>
<td>GT</td>
<td>Greater than</td>
<td>Z = 1 or Z = 1</td>
</tr>
<tr>
<td>0111</td>
<td>GE</td>
<td>Greater than or equal</td>
<td>N = V</td>
</tr>
<tr>
<td>1000</td>
<td>HE</td>
<td>Unsigned higher</td>
<td>C = 1 and Z = 0</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>Unsigned lower or equal</td>
<td>C = 0 or Z = 1</td>
</tr>
<tr>
<td>1010</td>
<td>GT</td>
<td>Signed greater than or equal</td>
<td>N = V</td>
</tr>
<tr>
<td>1011</td>
<td>GE</td>
<td>Signed greater than</td>
<td>Z = 1 or Z = 1</td>
</tr>
<tr>
<td>1100</td>
<td>LE</td>
<td>Signed less than</td>
<td>Z = 0 or Z = 0</td>
</tr>
<tr>
<td>1101</td>
<td>LT</td>
<td>Signed less than or equal</td>
<td>N = V</td>
</tr>
<tr>
<td>1110</td>
<td>EQ</td>
<td>Always (unconditional)</td>
<td>Always (unconditional)</td>
</tr>
<tr>
<td>1111</td>
<td>NE</td>
<td>Always (unconditional)</td>
<td>Always (unconditional)</td>
</tr>
</tbody>
</table>

IT blocks

- Conditional execution in C-M3 done in “IT” block
- IT [T|E]*3
- More on this later...

The ARM architecture “books” for this class

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
  !bne!done  // NE true when Z==0
  !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?

How are assembly files assembled?

• $ arm-none-eabi-as
  - Useful options
    • -mcpu
    • -mthumb
    • -O

$ arm-none-eabi-as -mcpu=cortex-a3 -mthumb example1.s -o example1.o

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++
What’s it all mean?

```assembly
.equ STACK_TOP, 0x20000000
.text /* Equates symbol to value */
.syntax unified
.thumb
.global _start
.type _start, SFunction
.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

```bash
all: arm-none-eabi-as -mcu=cortex-m3 -mthumb example1.s -o example1.o
    arm-none-eabi-id -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?

```assembly
.equ STACK_TOP, 0x20000000
.text
.syntax unified
.thumb
.global _start
.type _start, SFunction
.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
```

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

```
C files (.c)

Assembly files (.s)

Library object files (.o)

Linker script (.ld)

Executable image file

Binary program file (.bin)

Library object files (.o)

Disassembled Code (.lst)
```

Today...

Finish ARM assembly example from last time

Walk through 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).