ARM Architecture and Assembly Programming Intro

Instructors:
Dr. Phillip Jones
Announcements

• Lab 9: object detection: Give TAs Creative Team name & verify team
• Exam 3: Friday 7/6 (last day of class)
• Final Projects
  – Projects: Mandatory Demos Friday, 7/6 (11am – 3pm)
  – Reminder, Lab attendance is mandatory: -10 points from final project for each lab session you miss
• Homework 6: Due Sunday (7/1)
• Quiz 6 (Thursday, 6/28): Textbook readings below (in Chp2 and 4).
• Reading for the next few weeks
  – Textbook: Chapter 2.1-2.3, and 2.6.1-2.6.2
  – Textbook: Chapter 4.1 – 4.3
  – Assemble ARM instruction set manual:
    • Preface, Chapter 3, Chapter 4
  – ARM Procedure Call Standard:
    • Sections: 5, 7.1.1, 7.2.
Why use assembly programming?

• Full access to hardware features
  – Compiler limits a programmers access to the hardware features that the compiler writer decided to implement

• Writing time critical portions of code
  – Allows tight control over what the CPU is doing on every clock cycle

• Debugging
  – It is not uncommon when trying to debug odd system behavior to have to look at disassembled code
Why learn the ARM Hardware Architecture?

• Helps give intuition to why the assembly instructions were created the way they were.
• Help understand what special feature may be available for you to make use of.
Table 2.1 Development history of the ARM® MCUs family.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Bit Width</th>
<th>Cores Designed by ARM Holdings</th>
<th>Cores Designed by Third Parties</th>
<th>Cortex Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv1</td>
<td>32/26</td>
<td>ARM1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARMv2</td>
<td>32/26</td>
<td>ARM2, ARM3</td>
<td>Amber, STORM Open Soft Core</td>
<td></td>
</tr>
<tr>
<td>ARMv3</td>
<td>32</td>
<td>ARM6, ARM7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARMv4</td>
<td>32</td>
<td>ARM8</td>
<td>StrongARM, FA526</td>
<td></td>
</tr>
<tr>
<td>ARMv4T</td>
<td>32</td>
<td>ARM7TDMI, ARM9TDMI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARMv5</td>
<td>32</td>
<td>ARM7EJ, ARM9E, ARM10E</td>
<td>XScale, FA626TE, Ferocean, PJ1/Mohawk</td>
<td></td>
</tr>
<tr>
<td>ARMv6</td>
<td>32</td>
<td>ARM11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARMv6-M</td>
<td>32</td>
<td>ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1</td>
<td>Microcontroller</td>
<td></td>
</tr>
<tr>
<td>ARMv7-M</td>
<td>32</td>
<td>ARM Cortex-M3</td>
<td></td>
<td>Microcontroller</td>
</tr>
<tr>
<td>ARMv7E-M</td>
<td>32</td>
<td><strong>ARM Cortex-M4</strong></td>
<td></td>
<td>Microcontroller</td>
</tr>
<tr>
<td>ARMv7-R</td>
<td>32</td>
<td>ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7</td>
<td></td>
<td>Real-time</td>
</tr>
<tr>
<td>ARMv7-A</td>
<td>32</td>
<td>ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17</td>
<td>Krait, Scorpion, PJ4/Sheeva, Apple A6/A6X</td>
<td>Application</td>
</tr>
<tr>
<td>ARMv8-A</td>
<td>64/32</td>
<td>ARM Cortex-A53, ARM Cortex-A57</td>
<td>X-Gene, Denver, Apple A7 (Cyclone), K12</td>
<td>Application</td>
</tr>
<tr>
<td>ARMv8-R</td>
<td>32</td>
<td>No announcements yet</td>
<td></td>
<td>Real-time</td>
</tr>
</tbody>
</table>
ARM (Cortex-M4) Architecture Overview

- 32 bit processor
  - size of bus is 32 bits
  - size of registers is 32 bits
  - Size of instructions 16/32 bits
- RISC architecture
- Harvard architecture
  - separate data and instruction memory
- 203 instructions

Figure 2.1  The block diagram for a general MCU.

http://class.ece.iastate.edu/cpre288
Figure 2.1 The block diagram for a general MCU.
ARM: RISC CPU Architecture

• What is RISC?
  – Reduced Instruction Set Computing (with respect to CISC, Complex Instruction set Computing)

• Typical RISC
  – LD/Store based: ALU to memory transaction via registers
  – Most instruction are the same length
  – Typically many less instructions than a CISC architecture
  – Typically many more registers than CISC since Data must be moved into a register before it can be operated on
  – Low number of instruction typically makes hardware design simpler (as compared to CISC)

Ref: http://www.seas.upenn.edu/~palsetia/cit595s07/RISCvsCISC.pdf (Diana Palsetia)
## ARM: RISC vs CISC example

<table>
<thead>
<tr>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size / time</td>
<td>Size / time</td>
</tr>
<tr>
<td>1 byte, 1clk</td>
<td>1 byte, 1clk</td>
</tr>
<tr>
<td>1 byte, 1clk</td>
<td>1 byte, 1clk</td>
</tr>
<tr>
<td>4 byte, 30 clk</td>
<td>1 byte, 1 clk</td>
</tr>
</tbody>
</table>

### Instructions
- **CISC**: Instructions often variable length and variable time
  - **RISC**: Instruction typically constant length and time
    - Simpler hardware logic for decoding instructions (thus typically faster)

Ref: [http://www.seas.upenn.edu/~palsetia/cit595s07/RISCvsCISC.pdf](http://www.seas.upenn.edu/~palsetia/cit595s07/RISCvsCISC.pdf) (Diana Palsetia)
ARM CPU Core Summary

Instructions are 16-bit or 32-bit

Simple three-stage pipeline

Registers are 32-bit and addresses are 32-bit
Cortex-M4 Architecture

- 32 bit processor
  - size of data bus is 32 bits
  - size of registers is 32 bits
- Harvard architecture
- RISC architecture
- ~203 instructions

Figure 2.3   The functional block diagram for the ARM® Cortex®-M4 MCU.
Cortex-M4 Processor Architecture

- 32 bit processor
  - size of data bus is 32 bits
  - size of registers is 32 bits
- Harvard architecture
- RISC architecture
- ~203 instructions
• Program memory
  – Flash based: Program stays even if power turned off (non-volatile)
  – 16-bits wide, instructions are 16-bit or 32-bit wide.

• Data Memory
  – SRAM based: Data disappears if power is turned off (volatile)
  – 32-bits wide:
• Registers:
  – 32-bits wide
  – Directly accessible by ALU

• Data Memory:
  – 32-bit wide
  – Must use a register to move to/from the ALU
ARM: Memory Map organization

- Used for program codes & exception vector table
  - 0x00000000
  - 0x0003FFFF

- Used for program data
  - 0x20000000
  - 0x20007FFF

- Used for Peripherals
  - 0x40000000
  - 0x400FFFFF

- Used for Peripherals Including NVIC & Debug
  - 0xE0000000
  - 0xE0041FFF

- 256KB Flash ROM
- 32KB SRAM
- Peripheral Devices
- Private Peripheral Bus (PPB)

http://class.ece.iastate.edu/cpre288
Understanding Data

• Stack
  – Stores data related to function variables, function calls, parameters, return variables, etc.
  – Data on the stack can go “out of scope”, and is then automatically deallocated.
  – Starts at the top of the program’s data memory space, and addresses move down as more variables are allocated.

• Heap
  – Stores dynamically allocated data.
  – Dynamically allocated data usually calls the functions `alloc` or `malloc` (or uses `new` in C++) to allocate memory, and `free` to (or `delete` in C++) deallocate.
  – There’s no garbage collector!
  – Starts at bottom of program’s data memory space, and addresses move up as more variables are allocated.
ARM: Typical Logical Memory Layout

- Static Data (e.g. Global vars)
- Stack (e.g. local vars, function args & return value, return address)
- Heap (e.g. dynamically allocated memory: malloc, new)
- Code (Code may be in the same or a separate memory device)
Example Memory Layout

```c
static char[] greeting = "Hello world!";

main()
{
    int i;
    char bVal;

    LCD_init();
    LCD_PutString(greeting);
    ...
}
```

Memory Mapped I/O, Devices

- **I/O addresses**
- **Stack (grows down)**
- **Heap (grows up)**
- **Static Data**
- **Code**

http://class.ece.iastate.edu/cpre288
ARM: Typical Function Stack-Frame

- Stack is typically divided into a number of function stack-frames.
- Stack-Frame: Contains information associated with a function call
  - Function args, local vars, return address, preserved values

Stack

- Function Return Address
- Preserved Values
- Function Args
- Local Vars

Function 1 Frame

Function 2 Frame

Stack Base Address
Stack grows downward
Conventional program stack grows downwards: New items are put at the top, and the top grows down.
Function and Stack

Auto, local variables have their storage in stack

Why stack?

- The LIFO order matches perfectly with functions call/return order
  - LIFO: Last In, First Out
  - Function: Last called, first returned
- Efficient memory allocation and de-allocation
  - Allocation: Decrease SP (stack top): PUSH
  - De-allocation: Increase SP: POP
Function and Stack

Function Frame: Local storage for a function
What can be put in a stack frame?

- Function return address
- Parameter values
- Return value
- Local variables
- Saved register values
Example: Stack

• The following example shows the execution of a simple program (left) and the memory map of the stack (right)
Example: Stack

```c
void doNothing() {
    char c;
}

int main() {
    char x, y, z;
    short i;
    for (i = 0; i < 10; i++) {
        doNothing();
    }
    return 0;
}
```
Example: Stack

```c
void doNothing() {
    char c;
}

int main() {
    char x, y, z;
    short i;
    for (i = 0; i < 10; i++) {
        doNothing();
    }
    return 0;
}
```
void doNothing() {
    char c;
}

int main() {
    char x, y, z;
    short i;
    for (i = 0; i < 10; i++) {
        doNothing();
    }
    return 0;
}
```c
void doNothing() {
    char c;
}

int main() {
    char x, y, z;
    short i;
    for (i = 0; i < 10; i++) {
        doNothing();
    }
    return 0;
}
```
void doNothing() {
    char c;
}

int main() {
    char x, y, z;
    short i;
    for (i = 0; i < 10; i++) {
        doNothing();
    }
    return 0;
}
Example: Stack

```c
void doNothing() {
    char c;
}

int main() {
    char x, y, z;
    short i;
    for (i = 0; i < 10; i++) {
        doNothing();
    }
    return 0;
}
```

PC Address for line 9

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>x</td>
</tr>
<tr>
<td>999</td>
<td>y</td>
</tr>
<tr>
<td>998</td>
<td>z</td>
</tr>
<tr>
<td>997</td>
<td>i</td>
</tr>
<tr>
<td>996</td>
<td></td>
</tr>
<tr>
<td>995</td>
<td></td>
</tr>
<tr>
<td>994</td>
<td></td>
</tr>
<tr>
<td>993</td>
<td></td>
</tr>
<tr>
<td>....</td>
<td></td>
</tr>
<tr>
<td>....</td>
<td></td>
</tr>
</tbody>
</table>
void doNothing() {
    char c;
}

int main() {
    char x, y, z;
    short i;
    for (i = 0; i < 10; i++) {
        doNothing();
    }
    return 0;
}
General Purpose Register File

Figure 2.5   A structure block diagram of 21 registers in the Cortex®-M4 Core.

13 General Purpose Registers (R0 ~ R12)

Low Register

High Register

Stack Pointer Register

Link Register

Program Counter

Application PSR

Execution PSR

Interrupt PSR

Program Status Register

Interrupt Exception Mask Registers

Control Register

Main Stack Pointer (MSP)

Process Stack Pointer (PSP)
• 16 32-bit general purpose registers
  – Used for accessing SRAM
  – Used for storing function parameters
  – Used for instructions to execute operations on

• What is an 32-bit register.
  – Basically just 32 D-Flips connected together
STATUS REGISTER (SREG)
Status Register (SREG)

(a) Three individual registers – APSR, IPSR and EPSR.

(b) The combined register PSR.

Figure 2.6 Structure and bit functions in special registers.
Status Register (SREG): Common flags

• **Z: Zero flag**
  – Set to 1 when the result of an instruction is 0

• **C: Carry flag**
  – Set to 1 when the result of an instruction causes a carry to occur

• **N: Negative**
  – Set to 1 to indicate the result of an instruction is negative

• **V: Overflow**
  – Set to 1 to indicate the result of an instruction caused an overflow
Example ARM Assembly

```c
int x, a, b;
x = (a + b);
```

```asm
MOVW r4,address-of-a; get address for a
LDR r0,[r4]; get value of a
MOVW r4,address-of-b; get address for b
LDR r1,[r4]; get value of b
ADD r3,r0,r1; compute a+b
MOVW r4,address-of-x; get address for x
STR r3,[r4]; store value of x
```
How to Study Assembly

1. Get to know CPU registers
   Memorize rules for usage

2. Know basic types of Instruction
   Memory load and store
   Arithmetic/Logic
   Compare and branch
How to Study Assembly

3. Translate C statements
   - Memory accesses
   - Simple arithmetic statements
   - If statement
   - Loop statements

4. Translate C functions
   - Function Linkage
   - Making a function call
5. Interrupt System

- Principle of interrupt and exception
- Interrupt vector table
- Saving and restoring context
Challenges in learning Assembly

– Must understand how program works cycle by cycle
– Have to memorize some notations before fully understanding them
Announcements

• **Final Projects**
  – Project Teams: Demo Thursday 7/6
    • +5 bonus if Demoed by Wednesday 7/5
  – Reminder lab attendance is mandatory: -10 points from final project for each lab session you miss

• **Exam 2: In class Thursday 11/9**

• **Reading for remainder of semester**
  – Chapter 2.1-2.3, and 2.6.1-2.6.2
  – Chapter 4.1 – 4.3
  – Assemble ARM instruction set manual.
    • Preface, Chapter 3, Chapter 4
  – ARM Procedure Call Standard
    • Sections: 5, 7.1.1, 7.2.