ARM Architecture and Assembly Programming Intro

Instructors:
Dr. Phillip Jones
Announcements

• Exam 2: In class Tuesday 11/16
• Extra Office Hours:
  – Friday 11/13 : 6pm – 7pm
  – Monday 11/15: 6pm – 7pm
• HW 11: Due 11/21
• Reading for remainder of semester
  – Chapter 2.1-2.3, and 2.6.1-2.6.2 (Read)
  – Chapter 4.1 – 4.3 (Read)
  – ARM Instruction Set Summary (2-pages)
  – Assemble ARM instruction set manual (Scan).
    • Preface, Chapter 3, Chapter 4
  – ARM Procedure Call Standard (Scan)
    • Sections: 5, 7.1.1, 7.2. (~9pages)
ARM ARCHITECTURE OVERVIEW
Microcontroller / System-on-Chip (SoC)

Program Memory -> CPU

Interrupts

CPU

Data Memory

Devices

ADC
CFG | DATA | STATUS

UART
CFG | DATA | STATUS

Timers
CFG | DATA | STATUS

GPIO

GPIO_DATA

Port X (8-bits)

0 1 2 3 4 5 6 7

AFSEL

PCTL

http://class.ece.iastate.edu/cpre288
Microcontroller / System-on-Chip (SoC)

- Program Memory
- CPU
- Data Memory
- ADC
  - CFG | DATA | STATUS
- UART
  - CFG | DATA | STATUS
- Timers
  - CFG | DATA | STATUS
- GPIO
  - GPIO_DATA

Interrupts

Devices

Outside World

Port X (8-bits)

http://class.ece.iastate.edu/cpre288
Microcontroller / System-on-Chip (SoC)

Program Memory → CPU → Data Memory

Outside World
Microcontroller / System-on-Chip (SoC)

Microcontroller

Program Memory

CPU

Data Memory

Outside World

http://class.ece.iastate.edu/cpre288
Microcontroller / System-on-Chip (SoC)
Example ARM Assembly

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

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
```

http://class.ece.iastate.edu/cpre288
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.
ARM: Memory Map organization

- **Used for program codes & exception vector table**: 0x00000000
- **Used for program data**: 0x0003FFFF
- **Used for Peripherals**: 0x20000000
- **Used for Peripherals Including NVIC & Debug**: 0xE0000000

- 256KB Flash ROM: 0x00000000
- 32KB SRAM: 0x20000000
- Peripheral Devices: 0x40000000
- Private Peripheral Bus (PPB): 0xE0000000

CPU
ARM: Harvard Architecture

- **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:

http://class.ece.iastate.edu/cpre288
• Registers:
  – 32-bits wide
  – Directly accessible by ALU

• Data Memory:
  – 32-bit wide
  – Must use a register to move to/from the ALU
GENERAL PURPOSE & STATUS REGISTER (SREG)
General Purpose Register File

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

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

(b) The combined register PSR.

Figure 2.6  Structure and bit functions in special registers.
MEMORY ORGANIZATION
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)

Stack Pointer (SP): Address of top of the stack

Stack Base Address
Stack grows downward

Heap grows upward

0x0High address

Static Data

Heap

Data

Code Segment

Code

0x0000_0000
static char[] greeting = "Hello world!";

main()
{
  int i;
  char bVal;

  LCD_init();
  LCD_PutString(greeting);
  ...
}
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
Conventional program stack grows downwards: New items are put at the top, and the top grows down.
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

![Figure 1 - Stack Frame creation and destruction](image-url)
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)
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;
}
```

A stack diagram is also shown, illustrating the allocation and use of variables within the `for` loop.
Example: Stack

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

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

The figure shows a stack diagram with the following elements:
- The stack frame for the `main` function is shown.
- The stack frame grows from the bottom to the top.

The variables `x`, `y`, and `z` are pushed onto the stack, followed by the variable `i`. Each variable is pushed in order from the bottom to the top of the stack.
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
```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;
}
```

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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
```