DESN2000
(Computer Engineering)
ARM Assembly
Hasindu Gamaarachchi
ARM Assembly

• This is a crash course, well a “crash lecture”!
• Assumes prior learning of an assembly language
  • MIPS in COMP1521
• ARM is a Reduced Instruction Set Computer (RISC) architecture
  • Conceptually a lot of similarities to MIPS
  • And yeh, there are some differences
The upcoming slides are adapted from “Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C (Fourth Edition)” – Yifeng Zhu

• ARM Instruction set architecture - chapter 3
• For an in-depth understanding of ARM assembly programming – chapters 4-8
• Mixing C and assembly – chapter 10
### ARM Processor Register Bank

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0-R12</td>
<td>General purpose registers</td>
</tr>
<tr>
<td>R13 (SP)</td>
<td>Stack pointer (SP)</td>
</tr>
<tr>
<td>R14 (LR)</td>
<td>Link Register (LR)</td>
</tr>
<tr>
<td>R15 (PC)</td>
<td>Programme Counter (PC)</td>
</tr>
</tbody>
</table>

- Each register is 32 bits
- R0-R12: General purpose registers
- R13: Stack pointer (SP)
- R14: Link Register (LR)
- R15: Programme Counter (PC)
ADD r0, r2, r3  // r0 = r2 + r3

- mnemonic
- destination operand (r3)
- 1st source operand (r2)
- 2nd source operand (r0)
- comment
ARM Instruction Examples

ADD  r1, r2, r3    // r1 = r2 + r3
ADD  r1, r2, #4    // r1 = r2 + 4

MOV  r0, r1        // r0 = r1
MOV  r1, #5        // r1 = 5
ARM Assembly Instructions

- Arithmetic and logic
  - Add, Subtract, Multiply, Divide, Shift, Rotate

- Data movement
  - Load, Store, Move

- Compare and branch
  - Compare, If-then, branch,

- Miscellaneous
  - Breakpoints, wait for events, no operation
Arithmetic Instructions: Examples

- ADD $r0, r1, r2$ // $r0 = r1 + r2$
- ADC $r0, r1, r2$ // Add with carry, $r0 = r1 + r2 + carry$
- SUB $r0, r1, r2$ // $r0 = r1 - r2$
- SBC $r0, r1, r2$ // Subtract with borrow, $r0 = r1 - r2 - (1 - \text{carry})$
- MUL $r0, r1, r2$ // $r0 = r1 \times r2$, product limited to 32 bits
Logic Instructions: Examples

• **AND**  \( r_0, r_1, r_2 \) // Bitwise AND, \( r_0 = r_1 \text{ AND } r_2 \)

• **ORR**  \( r_0, r_1, r_2 \) // Bitwise OR, \( r_0 = r_1 \text{ OR } r_2 \)

• **EOR**  \( r_0, r_1, r_2 \) // Bitwise Exclusive OR, \( r_0 = r_1 \text{ EOR } r_2 \)

• **ORN**  \( r_0, r_1, r_2 \) // Bitwise OR NOT, \( r_0 = r_1 \text{ ORN } r_2 \)

• **BIC**  \( r_0, r_1, r_2 \) // Bit clear, \( r_0 = r_1 \& \sim r_2 \)

• **LSL**  \( r_0, r_1, r_2 \) // Logical shift left, \( r_0 = r_1 \ll r_2 \)

• **LSR**  \( r_0, r_1, r_2 \) // Logical shift right, \( r_0 = r_1 \gg r_2 \)

• **ROR**  \( r_0, r_1, r_2 \) // Rotate right, \( r_0 = r_1 \text{ rotate by } r_2 \text{ bits} \)
Data Movement Instructions: Examples

- **MOV** r1, r0  // r1 = r0
- **MOV** r1, #16  // r1 = 16
- **LDR** r1, =10000  // r1 = 10000

// assume r0 has the memory address
- **LDR** r1, [r0]  // r1 = Memory.word[r0]
- **LDR** r1, [r0,#8]  // r1 = Memory.word[r0+8]

; assume r0 has the memory address
- **STR** r1, [r0]  // Memory.word[r0] = r1
- **STR** r1, [r0,#8]  // Memory.word[r0+8] = r1
Compare and branch Instructions: Examples

**C Program**

```c
if (a == 1)
    b = 3;
else
    b = 4;
```

If-then-else

```plaintext
// r1 = a, r2 = b
CMP r1, #1  // compare a and 1
BNE else    // go to else if a ≠ 1
then:
    MOV r2, #3  // b = 3
B endif      // go to endif
else:
    MOV r2, #4  // b = 4
Endif:
```
For Loop

**C Program**

```c
int i;
int sum = 0;
for(i = 0; i < 10; i++){
    sum += i;
}
```

**MOV r0, #0 // i**
**MOV r1, #0 // sum**

**loop:**
**CMP r0, #10 // compare i and 10**
**BGE stop // end loop if i>=10**
**ADD r1, r1, r0 // sum += i**
**ADD r0, r0, #1 // i++**
**B loop // loop**

**stop:**
void foo(void);

int main(void{
    ...
    foo();
    ...
}

void foo(void) {
    ...
    return;
}

function call is referred to by terms such as procedure call and subroutine call.
• **NOP**  // No Operation
Function Calls – Calling Standard

• What is it?
  • Contract between a caller and callee

• Why need it?
  • Allows functions to be separately written, compiled, and assembled but work together
  • Allows C program call an assembly function, or vice versa

• Involves:
  • Argument passing
  • Return values
  • Backing up registers

• Calling standard is also known by terms like calling convention
Passing Arguments and Returning Value

Extra arguments are pushed to the stack by the caller. The caller is responsible to pop them out of the stack after the subroutine returns.

Subroutine

32-bit Argument 1
32-bit Argument 2
32-bit Argument 3
32-bit Argument 4
64-bit Argument 1
64-bit Argument 2
128-bit Argument
32-bit Return Value
64-bit Return Value
128-bit Return Value
Passing Arguments and Returning Value

```c
int32_t sum(uint8_t a8, int8_t b8, uint16_t c16, uint16_t d16);

s = sum(1, 2, 3, 4);
```

**Caller**

```assembly
MOVS r0, #1 // a8
MOVS r1, #2 // b8
MOVS r2, #3 // c16
MOVS r3, #4 // d16
BL sum
```

**Callee**

```assembly
Sum:
ADD r0, r0, r1 // a8 + b8
ADD r0, r0, r2 // add c16
ADD r0, r0, r3 // add d16
BX LR
```
Preserving registers

- Callee can freely modify R0, R1, R2, and R3
- If caller expects their values are retained, caller should push them onto the stack before calling the callee

- Caller expects these values are retained.
- If Callee modifies them, callee must restore their values upon leaving the function.
Preserving registers

<table>
<thead>
<tr>
<th>Caller Program</th>
<th>Subroutine/Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV r4, #100</td>
<td>foo:</td>
</tr>
<tr>
<td>...</td>
<td>PUSH {r4} // preserve r4</td>
</tr>
<tr>
<td>BL foo</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>MOV r4, #10 // foo changes r4</td>
</tr>
<tr>
<td>ADD r4, r4, #1</td>
<td>...</td>
</tr>
<tr>
<td>// r4 = 101, not 11</td>
<td>POP {r4} // Recover r4</td>
</tr>
<tr>
<td></td>
<td>BX LR</td>
</tr>
</tbody>
</table>

Caller expects callee does not modify r4!  
Callee should preserve r4!
Mixing C and Assembly

• Inline assembly

```c
// ...
// C code
__asm__ ( 
"MOV r0, #0x04800000 \n" 
"LDR r1, [r0, #0x14] \n" 
"BIC r1, r1, #1<<5 \n" 
"STR r1, [r0, #0x14] \n"
);
//... 
// C code
```
Mixing C and Assembly

- Calling as assembly function from C

<table>
<thead>
<tr>
<th>C Program (main.c)</th>
<th>Assembly Program (strlen.s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char str[25] = &quot;Hello!&quot;;</td>
<td>.global strlen</td>
</tr>
<tr>
<td>extern void int strlen(char* s);</td>
<td>strlen:</td>
</tr>
<tr>
<td>int main(void){</td>
<td>PUSH {r4} // preserve r4</td>
</tr>
<tr>
<td>int i;</td>
<td>MOV r4, #0 // initialize length</td>
</tr>
<tr>
<td>i = strlen(str);</td>
<td>loop:</td>
</tr>
<tr>
<td>while(1);</td>
<td>LDRB r1, [r0, r4] // r0 = string address</td>
</tr>
<tr>
<td>}</td>
<td>CBZ r1, exit // branch if zero</td>
</tr>
<tr>
<td></td>
<td>ADD r4, r4, #1 // length++</td>
</tr>
<tr>
<td></td>
<td>B loop // do it again</td>
</tr>
<tr>
<td></td>
<td>Exit:</td>
</tr>
<tr>
<td></td>
<td>MOV r0, r4 // place result in r0</td>
</tr>
<tr>
<td></td>
<td>POP {r4} // return</td>
</tr>
<tr>
<td></td>
<td>BX LR</td>
</tr>
</tbody>
</table>
Arithmetic and Logic Instructions

- **Shift**: LSL (logic shift left), LSR (logic shift right), ASR (arithmetic shift right), ROR (rotate right), RRX (rotate right with extend)
- **Logic**: AND (bitwise and), ORR (bitwise or), EOR (bitwise exclusive or), ORN (bitwise or not), MVN (move not)
- **Bit set/clear**: BFC (bit field clear), BFI (bit field insert), BIC (bit clear), CLZ (count leading zeroes)
- **Bit/byte reordering**: RBIT (reverse bit order in a word), REV (reverse byte order in a word), REV16 (reverse byte order in each half-word independently), REVSH (reverse byte order in each half-word independently)
- **Addition**: ADD, ADC (add with carry)
- **Subtraction**: SUB, RSB (reverse subtract), SBC (subtract with carry)
- **Multiplication**: MUL (multiply), MLA (multiply-accumulate), MLS (multiply-subtract), SMULL (signed long multiply-accumulate), SMLAL (signed long multiply-accumulate), UMULL (unsigned long multiply-subtract), UMLAL (unsigned long multiply-subtract)
- **Division**: SDIV (signed), UDIV (unsigned)
- **Saturation**: SSAT (signed), USAT (unsigned)
- **Sign extension**: SXTB (signed), SXTH, UXTB, UXTH
- **Bit field extract**: SBFX (signed), UBFX (unsigned)
Load/Store a Byte, Halfword, Word

\[ \text{LDRxxx R0, [R1]} \]
// Load data from memory into a \textbf{32-bit} register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Data Type</th>
<th>signed/unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>Load Word</td>
<td>uint32_t/int32_t</td>
<td>unsigned or signed int</td>
</tr>
<tr>
<td>LDRB</td>
<td>Load Byte</td>
<td>uint8_t</td>
<td>unsigned char</td>
</tr>
<tr>
<td>LDRH</td>
<td>Load Halfword</td>
<td>uint16_t</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>LDRSB</td>
<td>Load Signed Byte</td>
<td>int8_t</td>
<td>signed char</td>
</tr>
<tr>
<td>LDRSH</td>
<td>Load Signed Halfword</td>
<td>int16_t</td>
<td>signed short int</td>
</tr>
</tbody>
</table>

\[ \text{STRxxx R0, [R1]} \]
// Store data extracted from a \textbf{32-bit} register into memory

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Data Type</th>
<th>signed/unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR</td>
<td>Store Word</td>
<td>uint32_t/int32_t</td>
<td>unsigned or signed int</td>
</tr>
<tr>
<td>STRB</td>
<td>Store Lower Byte</td>
<td>uint8_t/int8_t</td>
<td>unsigned or signed char</td>
</tr>
<tr>
<td>STRH</td>
<td>Store Lower Halfword</td>
<td>uint16_t/int16_t</td>
<td>unsigned or signed short</td>
</tr>
</tbody>
</table>
Pre-index and Post-index

<table>
<thead>
<tr>
<th>Index Format</th>
<th>Example</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-index</td>
<td>LDR r1, [r0, #4]</td>
<td>r1 ← memory[r0 + 4], r0 is unchanged</td>
</tr>
<tr>
<td>Pre-index with update</td>
<td>LDR r1, [r0, #4]!</td>
<td>r1 ← memory[r0 + 4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r0 ← r0 + 4</td>
</tr>
<tr>
<td>Post-index</td>
<td>LDR r1, [r0], #4</td>
<td>r1 ← memory[r0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r0 ← r0 + 4</td>
</tr>
</tbody>
</table>

Offset range is -255 to +255
Unconditional Branch Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>label</td>
<td>Branch</td>
</tr>
<tr>
<td>BL</td>
<td>label</td>
<td>Branch with Link</td>
</tr>
<tr>
<td>BX</td>
<td>Rm</td>
<td>Branch indirect</td>
</tr>
</tbody>
</table>

- **B label**
  - perform a branch to label.

- **BL label**
  - copy the address of the next instruction into r14 (lr, the link register), and
  - perform a branch to label.

- **BX Rm**
  - branch to the address held in Rm
Condition Flags

Program Status Register (PSR)

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>V</th>
</tr>
</thead>
</table>

- **Negative** bit
  - \( N = 1 \) if most significant bit of result is 1

- **Zero** bit
  - \( Z = 1 \) if all bits of the result are 0

- **Carry** bit
  - For unsigned addition, \( C = 1 \) if carry takes place
  - For unsigned subtraction, \( C = 0 \) (carry = not borrow) if borrow takes place
  - For shift/rotation, \( C = \) last bit shifted out

- **Overflow** bit
  - \( V = 1 \) if adding 2 same-signed numbers produces a result with the opposite sign
    - Positive + Positive = Negative, or
    - Negative + negative = Positive
  - Non-arithmetic operations does not touch V bit, such as MOV, AND, LSL, MUL
# Branch Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Flags tested</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Conditional Branch</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ label</td>
<td>Branch if <strong>E</strong>Qual</td>
<td>Z = 1</td>
</tr>
<tr>
<td>BNE label</td>
<td>Branch if <strong>N</strong>ot <strong>E</strong>qual</td>
<td>Z = 0</td>
</tr>
<tr>
<td>BCS/BHS label</td>
<td>Branch if unsigned <strong>H</strong>igher or <strong>S</strong>ame</td>
<td>C = 1</td>
</tr>
<tr>
<td>BCC/BLO label</td>
<td>Branch if unsigned <strong>L</strong>ower</td>
<td>C = 0</td>
</tr>
<tr>
<td>BMI label</td>
<td>Branch if <strong>M</strong>inus (Negative)</td>
<td>N = 1</td>
</tr>
<tr>
<td>BPL label</td>
<td>Branch if <strong>P</strong>lus (Positive or Zero)</td>
<td>N = 0</td>
</tr>
<tr>
<td>BVS label</td>
<td>Branch if <strong>o</strong>verflow <strong>S</strong>et</td>
<td>V = 1</td>
</tr>
<tr>
<td>BVC label</td>
<td>Branch if <strong>o</strong>verflow <strong>C</strong>lear</td>
<td>V = 0</td>
</tr>
<tr>
<td>BHI label</td>
<td>Branch if unsigned <strong>H</strong>igher</td>
<td>C = 1 &amp; Z = 0</td>
</tr>
<tr>
<td>BLS label</td>
<td>Branch if unsigned <strong>L</strong>ower or <strong>S</strong>ame</td>
<td>C = 0 or Z = 1</td>
</tr>
<tr>
<td>BGE label</td>
<td>Branch if signed <strong>G</strong>reater or <strong>E</strong>qual</td>
<td>N = V</td>
</tr>
<tr>
<td>BLT label</td>
<td>Branch if signed <strong>L</strong>ess <strong>T</strong>han</td>
<td>N != V</td>
</tr>
<tr>
<td>BGT label</td>
<td>Branch if signed <strong>G</strong>reater <strong>T</strong>han</td>
<td>Z = 0 &amp; N = V</td>
</tr>
<tr>
<td>BLE label</td>
<td>Branch if signed <strong>L</strong>ess than or <strong>E</strong>qual</td>
<td>Z = 1 or N != V</td>
</tr>
</tbody>
</table>
# ARM Procedure Call Standard

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
<th>Subroutine Preserved</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>Argument 1 and return value</td>
<td>No</td>
<td>If return has 64 bits, then r0:r1 hold it. If argument 1 has 64 bits, r0:r1 hold it.</td>
</tr>
<tr>
<td>r1</td>
<td>Argument 2</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>Argument 3</td>
<td>No</td>
<td>If the return has 128 bits, r0-r3 hold it.</td>
</tr>
<tr>
<td>r3</td>
<td>Argument 4</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td>General-purpose V1</td>
<td>Yes</td>
<td>Variable register 1 holds a local variable.</td>
</tr>
<tr>
<td>r5</td>
<td>General-purpose V2</td>
<td>Yes</td>
<td>Variable register 2 holds a local variable.</td>
</tr>
<tr>
<td>r6</td>
<td>General-purpose V3</td>
<td>Yes</td>
<td>Variable register 3 holds a local variable.</td>
</tr>
<tr>
<td>r7</td>
<td>General-purpose V4</td>
<td>Yes</td>
<td>Variable register 4 holds a local variable.</td>
</tr>
<tr>
<td>r8</td>
<td>General-purpose V5</td>
<td>YES</td>
<td>Variable register 5 holds a local variable.</td>
</tr>
<tr>
<td>r9</td>
<td>Platform specific/V6</td>
<td>Yes/No</td>
<td>Usage is platform-dependent. Can be Variable register 6</td>
</tr>
<tr>
<td>r10</td>
<td>General-purpose V7</td>
<td>Yes</td>
<td>Variable register 7 holds a local variable.</td>
</tr>
<tr>
<td>r11</td>
<td>General-purpose V8</td>
<td>Yes</td>
<td>Variable register 8 holds a local variable.</td>
</tr>
<tr>
<td>r12 (IP)</td>
<td>Intra-procedure-call register</td>
<td>No</td>
<td>It holds intermediate values between a procedure and the sub-procedure it calls.</td>
</tr>
<tr>
<td>r13 (SP)</td>
<td>Stack pointer</td>
<td>Yes</td>
<td>SP has to be the same after a subroutine has completed.</td>
</tr>
<tr>
<td>r14 (LR)</td>
<td>Link register</td>
<td>No</td>
<td>LR does not have to contain the same value after a subroutine has completed.</td>
</tr>
<tr>
<td>r15 (PC)</td>
<td>Program counter</td>
<td>N/A</td>
<td>Do not directly change PC</td>
</tr>
</tbody>
</table>
ARM Cortex Instruction Set

• https://web.eece.maine.edu/~zhu/book/Appendix_B_Cortex_M3_M4_Instructions.pdf
• Following posts would be helpful those who want to know how and what immediate values are supported in ARM instructions
  • https://xlogicx.net/ARM_12-bit_Immediates_are_Too_High_Level.html
  • https://minhhua.com/arm_immed_encoding/index.html