ARM Assembly Programming

Computer Organization and Assembly Languages
Yung-Yu Chuang
2007/11/19

with slides by Peng-Sheng Chen

Introduction

- The ARM processor is very easy to program at the assembly level. (It is a RISC)
- We will learn ARM assembly programming at the user level and run it on a GBA emulator.

ARM programmer model

- The state of an ARM system is determined by the content of visible registers and memory.
- A user-mode program can see 15 32-bit general-purpose registers (R0-R14), program counter (PC) and CPSR.
- Instruction set defines the operations that can change the state.

Memory system

- Memory is a linear array of bytes addressed from 0 to $2^{32} - 1$
- Word, half-word, byte
- Little-endian

<table>
<thead>
<tr>
<th>Address (Hex)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>00</td>
</tr>
<tr>
<td>0x00000001</td>
<td>10</td>
</tr>
<tr>
<td>0x00000002</td>
<td>20</td>
</tr>
<tr>
<td>0x00000003</td>
<td>30</td>
</tr>
<tr>
<td>0x00000004</td>
<td>FF</td>
</tr>
<tr>
<td>0x00000005</td>
<td>FF</td>
</tr>
<tr>
<td>0x00000006</td>
<td>FF</td>
</tr>
<tr>
<td>0xFFF000000</td>
<td>00</td>
</tr>
<tr>
<td>0xFFF000001</td>
<td>00</td>
</tr>
<tr>
<td>0xFFF000002</td>
<td>00</td>
</tr>
<tr>
<td>0xFFF000003</td>
<td>00</td>
</tr>
<tr>
<td>0xFFF000004</td>
<td>00</td>
</tr>
<tr>
<td>0xFFF000005</td>
<td>00</td>
</tr>
<tr>
<td>0xFFF000006</td>
<td>00</td>
</tr>
</tbody>
</table>
**Byte ordering**

- **Big Endian**
  - Least significant byte has highest address
  - Word address 0x00000000
  - Value: 00102030
- **Little Endian**
  - Least significant byte has lowest address
  - Word address 0x00000000
  - Value: 30201000

**ARM programmer model**

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000001</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000002</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000003</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000004</td>
<td>FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000005</td>
<td>FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000006</td>
<td>FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000007</td>
<td>00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000001</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000002</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000003</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000004</td>
<td>FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000005</td>
<td>FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000006</td>
<td>FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000007</td>
<td>00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000001</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000002</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000003</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000004</td>
<td>FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000005</td>
<td>FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000006</td>
<td>FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000007</td>
<td>00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R12</th>
<th>R13</th>
<th>R14</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000001</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000002</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000003</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000004</td>
<td>FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000005</td>
<td>FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000006</td>
<td>FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000007</td>
<td>00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction set**

ARM instructions are all 32-bit long (except for Thumb mode). There are $2^{32}$ possible machine instructions. Fortunately, they are structured.

**Features of ARM instruction set**

- Load-store architecture
- 3-address instructions
- Conditional execution of every instruction
- Possible to load/store multiple register at once
- Possible to combine shift and ALU operations in a single instruction
Instruction set

MOV<cc><S> Rd, <operands>

MOVCS R0, R1 @ if carry is set
   @ then R0:=R1

MOVS R0, #0 @ R0:=0
   @ Z=1, N=0
   @ C, V unaffected

Data processing

• Arithmetic and logic operations
• General rules:
  - All operands are 32-bit, coming from registers or literals.
  - The result, if any, is 32-bit and placed in a register (with the exception for long multiply which produces a 64-bit result)
  - 3-address format

Arithmetic

• ADD R0, R1, R2 @ R0 = R1+R2
• ADC R0, R1, R2 @ R0 = R1+R2+C
• SUB R0, R1, R2 @ R0 = R1-R2
• SBC R0, R1, R2 @ R0 = R1-R2+C-1
• RSB R0, R1, R2 @ R0 = R2-R1
• RSC R0, R1, R2 @ R0 = R2-R1+C-1

Instruction set

• Data processing (Arithmetic and Logical)
• Data movement
• Flow control
**Bitwise logic**

- **AND** 
  \[ R0, R1, R2 \] \[ R0 = R1 \text{ and } R2 \]
- **ORR** 
  \[ R0, R1, R2 \] \[ R0 = R1 \text{ or } R2 \]
- **EOR** 
  \[ R0, R1, R2 \] \[ R0 = R1 \text{ xor } R2 \]
- **BIC** 
  \[ R0, R1, R2 \] \[ R0 = R1 \text{ and } (\sim R2) \]

**bit clear:** \( R2 \) is a mask identifying which bits of \( R1 \) will be cleared to zero

\[ R1=0x11111111 \quad R2=0x01100101 \]

\[ \text{BIC } R0, R1, R2 \]

\[ R0=0x10011010 \]

---

**Register movement**

- **MOV** 
  \[ R0, R2 \] \[ R0 = R2 \]
- **MVN** 
  \[ R0, R2 \] \[ R0 = \sim R2 \]

**move negated**

---

**Comparison**

- These instructions do not generate a result, but set condition code bits (\( N, Z, C, V \)) in CPSR. Often, a branch operation follows to change the program flow.
  
  **compare**
  - **CMP** 
    \[ R1, R2 \] \[ \text{set } cc \text{ on } R1-R2 \]
  
  **compare negated**
  - **CMN** 
    \[ R1, R2 \] \[ \text{set } cc \text{ on } R1+R2 \]
  
  **bit test**
  - **TST** 
    \[ R1, R2 \] \[ \text{set } cc \text{ on } R1 \text{ and } R2 \]
  
  **test equal**
  - **TEQ** 
    \[ R1, R2 \] \[ \text{set } cc \text{ on } R1 \text{ xor } R2 \]

---

**Addressing modes**

- **Register operands**
  - **ADD** 
    \[ R0, R1, R2 \]

- **Immediate operands**
  - A literal; most can be represented by \( (0..255) \times 2^n \) \( 0 < n < 12 \)

  \[ \text{ADD } R3, R3, \#1 \quad R3:=R3+1 \]

  \[ \text{AND } R8, R7, \#0xff \quad R8=R7[7:0] \]
Shifted register operands

- One operand to ALU is routed through the Barrel shifter. Thus, the operand can be modified before it is used. Useful for dealing with lists, table and other complex data structure. (similar to the displacement addressing mode in CISC.)

Logical shift left

```
MOV R0, R2, LSL #2 @ R0:=R2<<2  @ R2 unchanged
Example: 0...0 0011 0000
Before R2=0x00000030
After  R0=0x000000C0
```

Logical shift right

```
MOV R0, R2, LSR #2 @ R0:=R2>>2  @ R2 unchanged
Example: 0...0 0011 0000
Before R2=0x00000030
After  R0=0x0000000C
```

Arithmetic shift right

```
MOV R0, R2, ASR #2 @ R0:=R2>>2  @ R2 unchanged
Example: 1010 0...0 0011 0000
Before R2=0xA0000030
After  R0=0xE800000C
R2=0xA0000030
```
**Rotate right**

MOV R0, R2, ROR #2 @ R0 := R2 rotate
@ R2 unchanged

Example: 0...0 0011 0001
Before R2 = 0x00000031
After R0 = 0x4000000C
R2 = 0x00000031

**Rotate right extended**

MOV R0, R2, RRX @ R0 := R2 rotate
@ R2 unchanged

Example: 0...0 0011 0001
Before R2 = 0x00000031, C = 1
After R0 = 0x80000018, C = 1
R2 = 0x00000031

**Shifted register operands**

Shifted register operands

**Shifted register operands**

Shifted register operands
Shifted register operands

- It is possible to use a register to specify the number of bits to be shifted; only the bottom 8 bits of the register are significant.

```
ADD R0, R1, R2, LSL R3 @ R0 := R1 + R2 * 2^R3
```

Setting the condition codes

- Any data processing instruction can set the condition codes if the programmers wish it to.

### 64-bit addition

```
ADDS R2, R2, R0
ADC R3, R3, R1
```

### Multiplication

- **MUL** R0, R1, R2 @ R0 = (R1 * R2)[31:0]

- **Features**:
  - Second operand can’t be immediate
  - The result register must be different from the first operand
  - If S bit is set, C flag is meaningless

- See the reference manual (4.1.33)

### Multiply-accumulate

```
MLA R4, R3, R2, R1 @ R4 = R3 * R2 + R1
```

- Multiply with a constant can often be more efficiently implemented using shifted register operand

```
MOV R1, #35
MUL R2, R0, R1
or
ADD R0, R0, R0, LSL #2 @ R0' = 5 * R0
RSB R2, R0, R0, LSL #3 @ R2 = 7 * R0'
```
Data transfer instructions

- Move data between registers and memory
- Three basic forms
  - Single register load/store
  - Multiple register load/store
  - Single register swap: `SWP(B)`, atomic instruction for semaphore

Single register load/store

- The data items can be a 8-bit byte, 16-bit half-word or 32-bit word.

```
LDR  R0, [R1]  @ R0 := mem32[R1]
STR  R0, [R1]  @ mem32[R1] := R0
```

LDR, LDRH, LDRB for 32, 16, 8 bits
STR, STRH, STRB for 32, 16, 8 bits

Load an address into a register

- The pseudo instruction `ADR` loads a register with an address

```
table: .word 10
...
ADR  R0, table
```

Addressing modes

- Memory is addressed by a register and an offset.
  
  ```
  LDR  R0, [R1]  @ mem[R1]
  ```

- Three ways to specify offsets:
  - Constant
    
    ```
    LDR  R0, [R1, #4]  @ mem[R1+4]
    ```
  - Register
    
    ```
    LDR  R0, [R1, R2]  @ mem[R1+R2]
    ```
  - Scaled
    
    ```
    LDR  R0, [R1, R2, LSL #2]  @ mem[R1+4*R2]
    ```

- Assembler transfer pseudo instruction into a sequence of appropriate instructions
  
  ```
  sub r0, pc, #12
  ```
Addressing modes

- Pre-indexed addressing (LDR R0, [R1, #4])
  without a writeback
- Auto-indexing addressing (LDR R0, [R1, #4]!)
  calculation before accessing with a writeback
- Post-indexed addressing (LDR R0, [R1], #4)
  calculation after accessing with a writeback

Pre-indexed addressing

LDR R0, [R1, #4]  @ R0=mem[R1+4]
@ R1 unchanged

LDR R0, [R1, #4]!

Auto-indexing addressing

LDR R0, [R1, #4]!  @ R0=mem[R1+4]
@ R1=R1+4

LDR R0, [R1, #4]!

Post-indexed addressing

LDR R0, R1, #4  @ R0=mem[R1]
@ R1=R1+4

LDR R0,[R1, #4]!

No extra time; Fast;
Comparisons

- Pre-indexed addressing
  LDR R0, [R1, R2] @ R0=mem[R1+R2]
  @ R1 unchanged

- Auto-indexing addressing
  LDR R0, [R1, R2]! @ R0=mem[R1+R2]
  @ R1=R1+R2

- Post-indexed addressing
  LDR R0, [R1], R2 @ R0=mem[R1]
  @ R1=R1+R2

Application

ADR R1, table
loop: LDR R0, [R1]
ADD R1, R1, #4 @ operations on R0
...  
ADR R1, table
loop: LDR R0, [R1], #4
  @ operations on R0
  ...

Multiple register load/store

- Transfer large quantities of data more efficiently.
- Used for procedure entry and exit for saving and restoring workspace registers and the return address.

Multiple load/store register

LDM load multiple registers
STM store multiple registers

suffix meaning
IA increase after
IB increase before
DA decrease after
DB decrease before
Multiple load/store register

LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn−#<registers>*4+4
DB: addr:=Rn−#<registers>*4
For each Ri in <registers>
   IB: addr:=addr+4
   DB: addr:=addr−4
   Ri:=M[addr]
   IA: addr:=addr+4
   DA: addr:=addr−4
<!>: Rn:=addr

LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn−#<registers>*4+4
DB: addr:=Rn−#<registers>*4
For each Ri in <registers>
   IB: addr:=addr+4
   DB: addr:=addr−4
   Ri:=M[addr]
   IA: addr:=addr+4
   DA: addr:=addr−4
<!>: Rn:=addr
Multiple load/store register

LDMIA R0, {R1,R2,R3}
or
LDMIA R0, {R1-R3}

R1: 10  
R2: 20  
R3: 30  
R0: 0x10

data | addr  
---|---  
10 | 0x010  
20 | 0x014  
30 | 0x018  
40 | 0x01C  
50 | 0x020  
60 | 0x024

Multiple load/store register

LDMIA R0!, {R1,R2,R3}

R1: 10  
R2: 20  
R3: 30  
R0: 0x01C

data | addr  
---|---  
10 | 0x010  
20 | 0x014  
30 | 0x018  
40 | 0x01C  
50 | 0x020  
60 | 0x024

Multiple load/store register

LDMIB R0!, {R1,R2,R3}

R1: 20  
R2: 30  
R3: 40  
R0: 0x01C

data | addr  
---|---  
10 | 0x010  
20 | 0x014  
30 | 0x018  
40 | 0x01C  
50 | 0x020  
60 | 0x024

Multiple load/store register

LDMDA R0!, {R1,R2,R3}

R1: 40  
R2: 50  
R3: 60  
R0: 0x018

data | addr  
---|---  
10 | 0x010  
20 | 0x014  
30 | 0x018  
40 | 0x01C  
50 | 0x020  
60 | 0x024
**Multiple load/store register**

\[
\text{LDMDB} \ R0!, \ \{R1,R2,R3\}
\]

<table>
<thead>
<tr>
<th>R1: 30</th>
<th>R2: 40</th>
<th>R3: 50</th>
<th>R0: 0x018</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x010</td>
<td>10</td>
<td>0x014</td>
<td>20</td>
</tr>
<tr>
<td>0x018</td>
<td>30</td>
<td>0x01C</td>
<td>40</td>
</tr>
<tr>
<td>0x020</td>
<td>50</td>
<td>0x024</td>
<td>60</td>
</tr>
</tbody>
</table>

**Application**

- Copy a block of memory
  - R9: address of the source
  - R10: address of the destination
  - R11: end address of the source

\[
\text{loop:} \quad \text{LDMIA} \ R9!, \ \{R0-R7\} \\
\text{STMIA} \ R10!, \ \{R0-R7\} \\
\text{CMP} \ R9, \ R11 \\
\text{BNE} \ \text{loop}
\]

**Application**

- Stack (full: pointing to the last used; ascending: grow towards increasing memory addresses)

<table>
<thead>
<tr>
<th>mode</th>
<th>POP</th>
<th>=LDM</th>
<th>PUSH</th>
<th>=STM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full ascending (FA)</td>
<td>LDMFA</td>
<td>LDMDA</td>
<td>STMFA</td>
<td>STMIB</td>
</tr>
<tr>
<td>Full descending (FD)</td>
<td>LDMFD</td>
<td>LDMDA</td>
<td>STMFD</td>
<td>STMDB</td>
</tr>
<tr>
<td>Empty ascending (EA)</td>
<td>LDMEA</td>
<td>LDMDA</td>
<td>STMEA</td>
<td>STMIA</td>
</tr>
<tr>
<td>Empty descending (ED)</td>
<td>LDMED</td>
<td>LDMIB</td>
<td>STMED</td>
<td>STMDA</td>
</tr>
</tbody>
</table>

\[
\text{LDMFD} \ R13!, \ \{R2-R9\} \\
\ldots \ @ \text{modify} \ R2-R9 \\
\text{STMFD} \ R13!, \ \{R2-R9\}
\]

**Control flow instructions**

- Determine the instruction to be executed next
- Branch instruction
  - B label
    
    \[
    \ldots
    \]
  - label: ... 
- Conditional branches
  - MOV R0, #0 
  - \[
  \text{loop:} \quad \ldots \\
  \text{ADD} \ R0, \ R0, \ #1 \\
  \text{CMP} \ R0, \ #10 \\
  \text{BNE} \ \text{loop}
  \]
### Branch conditions

<table>
<thead>
<tr>
<th>Branch</th>
<th>Interpretation</th>
<th>Normal use</th>
</tr>
</thead>
<tbody>
<tr>
<td>B BAL</td>
<td>Unconditional</td>
<td>Always take this branch</td>
</tr>
<tr>
<td></td>
<td>Always</td>
<td>Always take this branch</td>
</tr>
<tr>
<td>BEQ</td>
<td>Equal</td>
<td>Comparison equal or zero result</td>
</tr>
<tr>
<td>BNE</td>
<td>Not equal</td>
<td>Comparison not equal or non-zero result</td>
</tr>
<tr>
<td>BPL</td>
<td>Plus</td>
<td>Result positive or zero</td>
</tr>
<tr>
<td>BMI</td>
<td>Minus</td>
<td>Result minus or negative</td>
</tr>
<tr>
<td>BCC</td>
<td>Carry clear</td>
<td>Arithmetic operation did not give carry-out</td>
</tr>
<tr>
<td>BLO</td>
<td>Lower</td>
<td>Unsigned comparison gave lower</td>
</tr>
<tr>
<td>BHS</td>
<td>Carry set Higher or same</td>
<td>Arithmetic operation gave carry-out</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unsigned comparison gave higher or same</td>
</tr>
<tr>
<td>BVS</td>
<td>Overflow clear</td>
<td>Signed integer operation: no overflow occurred</td>
</tr>
<tr>
<td>BVS</td>
<td>Overflow set</td>
<td>Signed integer operation: overflow occurred</td>
</tr>
<tr>
<td>BGT</td>
<td>Greater than</td>
<td>Signed integer comparison gave greater than</td>
</tr>
<tr>
<td>BGE</td>
<td>Greater or equal</td>
<td>Signed integer comparison gave greater or equal</td>
</tr>
<tr>
<td>BLT</td>
<td>Less than</td>
<td>Signed integer comparison gave less than</td>
</tr>
<tr>
<td>BLE</td>
<td>Less or equal</td>
<td>Signed integer comparison gave less than or equal</td>
</tr>
<tr>
<td>BHI</td>
<td>Higher</td>
<td>Unsigned comparison gave higher</td>
</tr>
<tr>
<td>BLS</td>
<td>Lower or same</td>
<td>Unsigned comparison gave lower or same</td>
</tr>
</tbody>
</table>

### Conditional execution

- Almost all ARM instructions have a condition field which allows it to be executed conditionally.

\[
\text{movcs} \ R0, R1
\]

### Branch and link

- BL instruction save the return address to \( R14 \) (\( lr \))

\[
\begin{align*}
\text{BL} & \quad \text{sub} \quad @ \text{call sub} \\
\text{CMP} & \quad R1, \ #5 \quad @ \text{return to here} \\
\text{MOVEQ} & \quad R1, \ #0 \\
\end{align*}
\]

- \( \text{sub} \): \quad @ \text{sub entry point}

\[
\begin{align*}
\text{MOV} & \quad PC, \ LR \quad @ \text{return}
\end{align*}
\]

- use stack to save/restore the return address and registers

\[
\begin{align*}
\text{sub1:} & \quad \text{STMFD} \ R13!, \ \{R0-R2,R14\} \\
\text{BL} & \quad \text{sub2} \\
\text{LDMFD} & \quad R13!, \ \{R0-R2,PC\}
\end{align*}
\]

\[
\begin{align*}
\text{sub2:} & \quad \ldots \\
\text{MOV} & \quad PC, \ LR
\end{align*}
\]
Conditional execution

```
CMP   R0, #5
BEQ   bypass @ if (R0!=5) {
    ADD   R1, R1, R0 @ R1=R1+R0-R2
    SUB   R1, R1, R2 @ }

bypass: ...
```

Rule of thumb: if the conditional sequence is three instructions or less, it is better to use conditional execution than a branch.

```
CMP   R0, #5 smaller and faster
ADDNE R1, R1, R0
SUBNE R1, R1, R2
```

Conditional execution

```
if ((R0==R1) && (R2==R3)) R4++
```

```
CMP   R0, R1
BNE   skip
CMP   R2, R3
BNE   skip
ADD   R4, R4, #1
```

skip: ...

```
CMP   R0, R1
CMPEQ R2, R3
ADDEQ R4, R4, #1
```

Instruction set

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDC</td>
<td>Add with Carry</td>
<td>NINV</td>
<td>Logical NOT</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>ORR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
<td>ESRB</td>
<td>Reverse Subtract</td>
</tr>
<tr>
<td>BNE</td>
<td>Branch on Condition</td>
<td>SBC</td>
<td>Subtract with Carry</td>
</tr>
<tr>
<td>BLX</td>
<td>Branch Link</td>
<td>SMULL</td>
<td>Multiply Signed Long</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>STM</td>
<td>Store Multiple</td>
</tr>
<tr>
<td>EOR</td>
<td>Exclusive OR</td>
<td>STR</td>
<td>Store Register (Word)</td>
</tr>
<tr>
<td>LDM</td>
<td>Load Multiple</td>
<td>SUB</td>
<td>Subtract</td>
</tr>
<tr>
<td>LDR</td>
<td>Load Register (Word)</td>
<td>SWI</td>
<td>Software Interrupt</td>
</tr>
<tr>
<td>LDRB</td>
<td>Load Register (Byte)</td>
<td>SWP</td>
<td>Swap Word Value</td>
</tr>
<tr>
<td>MLA</td>
<td>Multiply Accumulate</td>
<td>SWPB</td>
<td>Swap Byte Value</td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
<td>TEQ</td>
<td>Test Equivalence</td>
</tr>
<tr>
<td>MRS</td>
<td>Load SPSR or CPSR</td>
<td>TST</td>
<td>Test</td>
</tr>
<tr>
<td>MSR</td>
<td>Store to SPSR or CPSR</td>
<td>UMLAL</td>
<td>Mult Accum Signed Long</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>UMSULL</td>
<td>Multiply Signed Long</td>
</tr>
</tbody>
</table>

ARM assembly program

```
main:  
LDR   R1, value   @ load value
STR   R1, result
SWI   #11

value: .word 0x0000C123
result: .word 0
```
Shift left one bit

```
ADR R1, value
MOV R1, R1, LSL #0x1
STR R1, result
SWI #11
value: .word 4242
result: .word 0
```

Add two numbers

```
main:
ADR R1, value1
ADR R2, value2
ADD R1, R1, R2
STR R1, result
SWI #11
value1: .word 0x00000001
value2: .word 0x00000002
result: .word 0
```

64-bit addition

```
ADR R0, value1
LDR R1, [R0]  \[0x00000001, 0xF0000000\]
LDR R2, [R0, #4]  \[0x00000000, 0x10000000\]
ADR R0, value2
LDR R3, [R0]
LDR R4, [R0, #4]
ADDS R6, R2, R4
ADC R5, R1, R3
STR R5, [R0]  \[R5\]
STR R6, [R0, #4]  \[R6\]
SWI #11
value1: .word 0x00000001, 0xF0000000
value2: .word 0x00000000, 0x10000000
result: .word 0
```

Loops

- For loops
  ```
  for (i-0; i<10; i++) {a[i]=0;}
  ```

```
MOV R1, #0
ADR R2, A
MOV R0, #0
LOOP: CMP R0, #10
      BGE EXIT
      STR R1, [R2, R0, LSL #2]
      ADD R0, R0, #1
      B LOOP
EXIT: ..
```
Loops

• While loops

```
    LOOP: ... ; evaluate expression
    BEQ EXIT
    ... ; loop body
    B LOOP
    EXIT: ...
```

Find larger of two numbers

```
ADR R1, value1
ADR R2, value2
CMP R1, R2
BHI Done
MOV R1, R2
Done:
STR R1, result
SWI #11
value1: .word 4
value2: .word 9
result: .word 0
```

GCD

```
int gcd (int I, int j)
{
    while (i!=j)
    {
        if (i>j)
            i -= j;
        else
            j -= i;
    }
}
```

GCD

```
Loop:  CMP   R1, R2
       SUBGT R1, R1, R2
       SUBLT R2, R2, R1
       BNE   loop
```
Count negatives

; count the number of negatives in
; an array DATA of length LENGTH

ADR   R0, DATA  @ R0 addr
EOR   R1, R1, R1  @ R1 count
LDR   R2, Length  @ R2 index
CMP   R2, #0
BEQ   Done

Count negatives

loop:
    LDR   R3, [R0]
    CMP   R3, #0
    BPL   looptest
    ADD   R1, R1, #1  @ it’s neg.

looptest:
    ADD   R0, R0, #4
    SUBS  R2, R2, #1
    BNE   loop

Subroutines

• Passing parameters in registers
  Assume that we have three parameters
  BufferLen, BufferA, BufferB to pass into
  a subroutine

  ADR   R0, BufferLen
  ADR   R1, BufferA
  ADR   R2, BufferB
  BL Subr

Passing parameters using stacks

• Caller

MOV   R0, #BufferLen
STR   R0, [SP, #-4]!
MOV   R0, =BufferA
STR   R0, [SP, #-4]!
MOV   R0, =BufferB
STR   R0, [SP, #-4]!
BL    Subr
Passing parameters using stacks

- Callee
  Subr
  STMDB SP, {R0,R1,R2,R13,R14}
  LDR   R2, [SP, #0]
  LDR   R1, [SP, #4]
  LDR   R0, [SP, #8]
  ...
  LDMDB SP, {R0,R1,R2,R13,R14}
  MOV   PC, LR

Review

- ARM architecture
- ARM programmer model
- ARM instruction set
- ARM assembly programming

ARM programmer model

0x00000000 00
0x00000001 10
0x00000002 20
0x00000003 30
0x00000004 FF
0x00000005 FF
0x00000006 FF
0xFFF80000 00
0xFFF80001 00
0xFFF80002 00
0xFFF80003 00
0xFFF80004 00
0xFFF80005 00
0xFFF80006 00
### Instruction set

<table>
<thead>
<tr>
<th>Operation</th>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Operation</th>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add with Carry</td>
<td></td>
<td>NW</td>
<td>Logical NOT</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td></td>
<td>OR</td>
<td>Logical OR</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
<td></td>
<td>RSB</td>
<td>Reverse Subtract</td>
<td></td>
</tr>
<tr>
<td>BAL</td>
<td>Unconditional Branch</td>
<td></td>
<td>RSC</td>
<td>Reverse Subtract with Carry</td>
<td></td>
</tr>
<tr>
<td>B.(cc)</td>
<td>Branch on Condition</td>
<td></td>
<td>SBC</td>
<td>Subtract with Carry</td>
<td></td>
</tr>
<tr>
<td>BIC</td>
<td>Bit Clear</td>
<td></td>
<td>SMALL</td>
<td>Mult Accum Signed Long</td>
<td></td>
</tr>
<tr>
<td>BLAL</td>
<td>Unconditional Branch and Link</td>
<td></td>
<td>SMULL</td>
<td>Multiply Signed Long</td>
<td></td>
</tr>
<tr>
<td>BL.(cc)</td>
<td>Conditional Branch and Link</td>
<td></td>
<td>STM</td>
<td>Store Multiple</td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td></td>
<td>STR</td>
<td>Store Register (Word)</td>
<td></td>
</tr>
<tr>
<td>EOR</td>
<td>Exclusive OR</td>
<td></td>
<td>STRB</td>
<td>Store Register (Byte)</td>
<td></td>
</tr>
<tr>
<td>LDM</td>
<td>Load Multiple</td>
<td></td>
<td>SUB</td>
<td>Subtract</td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td>Load Register (Word)</td>
<td></td>
<td>SWI</td>
<td>Software Interrupt</td>
<td></td>
</tr>
<tr>
<td>LDRB</td>
<td>Load Register (Byte)</td>
<td></td>
<td>SWP</td>
<td>Swap Word Value</td>
<td></td>
</tr>
<tr>
<td>MLA</td>
<td>Multiply Accumulate</td>
<td></td>
<td>SWPB</td>
<td>Swap Byte Value</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
<td></td>
<td>TST</td>
<td>Text</td>
<td></td>
</tr>
<tr>
<td>MRS</td>
<td>Load SPSR or CPSR</td>
<td></td>
<td>UMLAL</td>
<td>Mult Accum Unsigned Long</td>
<td></td>
</tr>
<tr>
<td>MSR</td>
<td>Store to SPSR or CPSR</td>
<td></td>
<td>UMULL</td>
<td>Multiply Unsigned Long</td>
<td></td>
</tr>
</tbody>
</table>

### References

- Peter Knaggs and Stephen Welsh, *ARM: Assembly Language Programming.*
- Peter Cockerell, *ARM Assembly Language Programming.*