ARM Assembly Language

Introduction to ARM Basic Instruction Set
Microprocessors and Microcontrollers Course
Isfahan University of Technology, Dec. 2010
Main References

• *The ARM Architecture*
  • Presentation By ARM company itself
• *ARM Assembly Programming*
  • Presentation By Mr. Peng-Sheng Chen
Data Sizes and Instruction Set

- The ARM is a 32-bit architecture.

- When used in relation to the ARM:
  - **Byte** means 8 bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)

- Most ARM’s implement two instruction sets
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set

- Jazelle cores can also execute Java bytecode
Processor Modes

- The ARM has seven basic operating modes:
  - **User**: unprivileged mode under which most tasks run
  - **FIQ**: entered when a high priority (fast) interrupt is raised
  - **IRQ**: entered when a low priority (normal) interrupt is raised
  - **Supervisor**: entered on reset and when a Software Interrupt instruction is executed
  - **Abort**: used to handle memory access violations
  - **Undef**: used to handle undefined instructions
  - **System**: privileged mode using the same registers as user mode
ARM State & Thumb State

- ARM instruction set has two modes
  - ARM state: instructions are 32Bits
    - 16 registers are accessible
  - Thumb state: instructions are 16Bits
    - 8 registers are accessible
ARM Registers

- 16 Registers, 32Bits each
- R0 – R12
  - General purpose registers
- R13
  - Stack Pointer
- R14
  - Subroutine Link Register (LR)
  - Stores return address of subroutine
  - R14 stores a copy of R15 when BL instruction (Branch with Link) occurs
- R15
  - Program Counter
  - R15[1:0] always zero in ARM state
  - R15[0] always zero in Thumb State
ARM Registers

- CPSR
  - Current Program Status Register
  - Contains condition code flags
- SPSR
  - Saved Program Status Register
  - A copy of CPSR
ARM Register Set

Current Visible Registers

Abort Mode

Banked out Registers

User | FIQ | IRQ | SVC | Undef
--- | --- | --- | --- | ---
| r8 | r13 (sp) | r13 (sp) | r13 (sp) |
| r9 | r14 (lr) | r14 (lr) | r14 (lr) |
| r10 | r13 (sp) | r13 (sp) | r13 (sp) |
| r11 | r14 (lr) | r14 (lr) | r14 (lr) |
| r12 | r13 (sp) | r13 (sp) | r13 (sp) |
| r13 (sp) | r14 (lr) | r14 (lr) | r14 (lr) |
| cpsr | spsr | spsr | spsr |

Microprocessors & Microcontrollers - Mohammad Sadegh Sadri
ARM Registers

**User mode**
- r0-r7
- r15
- cpsr

**FIQ**
- User mode
- r0-r7, r15, and cpsr
- r8

**IRQ**
- User mode
- r0-r12, r15, and cpsr
- r13 (sp)
- r14 (lr)

**SVC**
- User mode
- r0-r12, r15, and cpsr
- r13 (sp)
- r14 (lr)

**Undf**
- User mode
- r0-r12, r15, and cpsr
- r13 (sp)
- r14 (lr)

**Abort**
- User mode
- r0-r12, r15, and cpsr
- r13 (sp)
- r14 (lr)

**Thumb state**
- Low registers
- High registers

**Note:** System mode uses the User mode register set
CPSR Register

- Condition code flags
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation overflowed

- J bit
  - Architecture 5TEJ only
  - J = 1: Processor in Jazelle state

- Interrupt Disable bits.
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.

- T Bit
  - Architecture xT only
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state

- Mode bits
  - Specify the processor mode
Exception Handling

- When an exception occurs, the ARM:
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - Change to ARM state
    - Change to exception mode
    - Disable interrupts (if appropriate)
  - Stores the return address in LR_<mode>
  - Sets PC to vector address
- To return, exception handler needs to:
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>

This can only be done in ARM state.

<table>
<thead>
<tr>
<th>Vector table can be at</th>
<th>0xFFFF0000 on ARM720T and on ARM9/10 family devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIQ</td>
<td>0x1C</td>
</tr>
<tr>
<td>IRQ</td>
<td>0x18</td>
</tr>
<tr>
<td>(Reserved)</td>
<td>0x14</td>
</tr>
<tr>
<td>Data Abort</td>
<td>0x10</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>0x0C</td>
</tr>
<tr>
<td>Software Interrupt</td>
<td>0x08</td>
</tr>
<tr>
<td>Undefined Instruction</td>
<td>0x04</td>
</tr>
<tr>
<td>Reset</td>
<td>0x00</td>
</tr>
</tbody>
</table>
Primary Assembly Language Programming for ARM

ARM INSTRUCTION SET
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
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

MOVVS R0, #0 @ R0:=0
@ Z=1, N=0
@ C, V unaffected
Instruction set

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

- **AND** R0, R1, R2 @ R0 = R1 and R2
- **ORR** R0, R1, R2 @ R0 = R1 or R2
- **EOR** R0, R1, R2 @ R0 = R1 xor R2
- **BIC** R0, R1, R2 @ R0 = R1 and (~R2)

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

R1=0x11111111  R2=0x01100101

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

R0=0x10011010
Register movement

- `MOV R0, R2 @ R0 = R2`
- `MVN R0, R2 @ R0 = ~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.

- **CMP R1, R2** @ set cc on R1-R2
  
  compare

- **CMN R1, R2** @ set cc on R1+R2
  
  compare negated

- **TST R1, R2** @ set cc on R1 and R2
  
  bit test

- **TEQ R1, R2** @ set cc on R1 xor R2
  
  test equal
Addressing modes

• Register operands

  ADD    R0, R1, R2

• Immediate operands

  a literal;

  ADD    R3, R3, #1   @ R3:=R3+1
  AND    R8, R7, #0xff @ R8=R7[7:0]

  a hexadecimal literal
  This is assembler dependent syntax.
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
R2 = 0x00000030
Logical shift right

MOV R0, R2, LSR #2 @ R0 := R2 >> 2
@ R2 unchanged

Example: 0...0 0011 0000
Before R2 = 0x00000030
After R0 = 0x0000000C
R2 = 0x00000030
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 Operation Examples**

- **LSL #5**
  - Original Value: 00000
  - Shifted Value: 00000

- **ASR #5, positive operand**
  - Original Value: 00000
  - Shifted Value: 00000 0

- **LSR #5**
  - Original Value: 00000
  - Shifted Value: 00000

- **ASR #5, negative operand**
  - Original Value: 1
  - Shifted Value: 1 1111 1

**Microprocessors & Microcontrollers - Mohammad Sadegh Sadri**
Shifted register operands

ROR #5

RRX
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

**ADDX**  R2, R2, R0
**ADC**   R3, R3, R1

\[
\begin{array}{c}
R1 \quad R0 \\
+ \\
R3 \quad R2 \\
\hline \\
R3 \quad R2
\end{array}
\]
Multiplication

- **MUL** R0, R1, R2 @ R0 = (R1xR2)\[31:0\]

- 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)
Multiplication

- Multiply-accumulate
  `MLA R4, R3, R2, R1 @ R4 = R3xR2+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’=5xR0`
  `RSB R2, R0, R0, LSL #3 @ R2 =7xR0’`
Data transfer instructions

- Move data between registers and memory
- Three basic forms
  - Single register load/store
  - Multiple register load/store
  - Single register swap: $\text{SWP} (B)$, 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.

\[
\begin{align*}
\text{LDR} & \quad R0, [R1] \quad R0 := \text{mem}_{32}[R1] \\
\text{STR} & \quad R0, [R1] \quad \text{mem}_{32}[R1] := R0 \\
\text{LDR, LDRH, LDRB} & \quad \text{for } 32, 16, 8 \text{ bits} \\
\text{STR, STRH, STRB} & \quad \text{for } 32, 16, 8 \text{ bits}
\end{align*}
\]
Load an address into a register

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

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

• Assembler transfer pseudo instruction into a sequence of appropriate instructions

```assembly
sub r0, pc, #12
```
Addressing modes

- Memory is addressed by a register and an offset.
  \[ \text{LDR R0, [R1] @ mem[R1]} \]

- Three ways to specify offsets:
  - Constant
    \[ \text{LDR R0, [R1, #4] @ mem[R1+4]} \]
  - Register
    \[ \text{LDR R0, [R1, R2] @ mem[R1+R2]} \]
  - Scaled
    \[ \text{LDR R0, [R1, R2, LSL #2] @ mem[R1+4*R2]} \]
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, ]
Auto-indexing addressing

LDR  R0, [R1, #4]!  @  R0=mem[R1+4]
@  R1=R1+4
No extra time; Fast;

LDR  R0, [R1, ]!  

R1

R0

Microprocessors & Microcontrollers - Mohammad Sadri

41
Post-indexed addressing

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

LDR R0, [R1]

R1

+}

R0
Comparisons

• Pre-indexed addressing
  \[ \text{LDR } R0, [R1, R2] \text{ @ } R0=\text{mem}[R1+R2] \text{ @ } R1 \text{ unchanged} \]

• Auto-indexing addressing
  \[ \text{LDR } R0, [R1, R2]! \text{ @ } R0=\text{mem}[R1+R2] \text{ @ } R1=R1+R2 \]

• Post-indexed addressing
  \[ \text{LDR } R0, [R1], R2 \text{ @ } R0=\text{mem}[R1] \text{ @ } R1=R1+R2 \]
Application

loop:

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

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

Microprocessors & Microcontrollers - Mohammad Sadegh Sadri

44
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.

Registers are arranged in increasing order; see manual.

```
LDMIA  R1, {R0, R2, R5} @ R0 = mem[R1]
        @ R2 = mem[r1+4]
        @ R5 = mem[r1+8]
```
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
DA: addr:=Rn
DB: addr:=Rn

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

Rn

R1
R2
R3
Multiple load/store register

LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn
DA: addr:=Rn
DB: addr:=Rn
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

LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn
DA: addr:=Rn
DB: addr:=Rn
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

LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn
DA: addr:=Rn
DB: addr:=Rn
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

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x010</td>
<td>10</td>
</tr>
<tr>
<td>0x014</td>
<td>20</td>
</tr>
<tr>
<td>0x018</td>
<td>30</td>
</tr>
<tr>
<td>0x01C</td>
<td>40</td>
</tr>
<tr>
<td>0x020</td>
<td>50</td>
</tr>
<tr>
<td>0x024</td>
<td>60</td>
</tr>
</tbody>
</table>
Multiple load/store register

LDMIA R0!, {R1,R2,R3}

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x010</td>
<td>10</td>
</tr>
<tr>
<td>0x014</td>
<td>20</td>
</tr>
<tr>
<td>0x018</td>
<td>30</td>
</tr>
<tr>
<td>0x01C</td>
<td>40</td>
</tr>
<tr>
<td>0x020</td>
<td>50</td>
</tr>
<tr>
<td>0x024</td>
<td>60</td>
</tr>
</tbody>
</table>

R1: 10  
R2: 20  
R3: 30  
R0: 0x01C
Multiple load/store register

LDMIB R0!, {R1,R2,R3}

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

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x010</td>
<td>10</td>
</tr>
<tr>
<td>0x014</td>
<td>20</td>
</tr>
<tr>
<td>0x018</td>
<td>30</td>
</tr>
<tr>
<td>0x01C</td>
<td>40</td>
</tr>
<tr>
<td>0x020</td>
<td>50</td>
</tr>
<tr>
<td>0x024</td>
<td>60</td>
</tr>
</tbody>
</table>
Multiple load/store register

LDM DA R0!, {R1, R2, R3}

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

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x010</td>
<td>10</td>
</tr>
<tr>
<td>0x014</td>
<td>20</td>
</tr>
<tr>
<td>0x018</td>
<td>30</td>
</tr>
<tr>
<td>0x01C</td>
<td>40</td>
</tr>
<tr>
<td>0x020</td>
<td>50</td>
</tr>
<tr>
<td>0x024</td>
<td>60</td>
</tr>
</tbody>
</table>
Multiple load/store register

LDMDB R0!, {R1,R2,R3}

R1: 30
R2: 40
R3: 50
R0: 0x018

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x010</td>
<td>10</td>
</tr>
<tr>
<td>0x014</td>
<td>20</td>
</tr>
<tr>
<td>0x018</td>
<td>30</td>
</tr>
<tr>
<td>0x01C</td>
<td>40</td>
</tr>
<tr>
<td>0x020</td>
<td>50</td>
</tr>
<tr>
<td>0x024</td>
<td>60</td>
</tr>
</tbody>
</table>
Application

• Copy a block of memory (32bytes aligned!)
  • R9: address of the source
  • R10: address of the destination
  • R11: end address of the source

``` Assembly
loop:  LDMIA R9!, {R0-R7}
       STMIA R10!, {R0-R7}
       CMP   R9, R11
       BNE   loop
```
Control flow instructions

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

<table>
<thead>
<tr>
<th>Branch</th>
<th>Interpretation</th>
<th>Normal uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>B BAL</td>
<td>Unconditional Always</td>
<td>Always take this branch</td>
</tr>
<tr>
<td></td>
<td></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 Lower</td>
<td>Arithmetic operation did not give carry-out</td>
</tr>
<tr>
<td>BLO</td>
<td>Carry set Higher or same</td>
<td>Unsigned comparison gave lower</td>
</tr>
<tr>
<td>BCS</td>
<td>Carry set Higher or same</td>
<td>Arithmetic operation gave carry-out</td>
</tr>
<tr>
<td>BHS</td>
<td>Carry set Higher or same</td>
<td>Unsigned comparison gave higher or same</td>
</tr>
<tr>
<td>BVC</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>
Branch and link

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

```assembly
BL    sub     @ call sub
CMP   R1, #5  @ return to here
MOVEQ R1, #0
...
sub:... @ sub entry point
...
MOV   PC, LR @ return
```
Application

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

<table>
<thead>
<tr>
<th>mode</th>
<th>LDM (POP)</th>
<th>STM (PUSH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascending</td>
<td>LDMDA</td>
<td>STMIB</td>
</tr>
<tr>
<td>descending</td>
<td>LDMIA</td>
<td>STMDB</td>
</tr>
<tr>
<td>Ascending</td>
<td>LDMDB</td>
<td>STMIA</td>
</tr>
<tr>
<td>descending</td>
<td>LDMIB</td>
<td>STMDA</td>
</tr>
</tbody>
</table>
### Application

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

<table>
<thead>
<tr>
<th>mode</th>
<th>LDM (POP)</th>
<th>STM (PUSH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full ascending (FA)</td>
<td>LDMDA</td>
<td>STMIB</td>
</tr>
<tr>
<td>Full descending (FD)</td>
<td>LDMIA</td>
<td>STMDB</td>
</tr>
<tr>
<td>Empty ascending (EA)</td>
<td>LDMDB</td>
<td>STMIA</td>
</tr>
<tr>
<td>Empty descending (ED)</td>
<td>LDMIB</td>
<td>STMDA</td>
</tr>
</tbody>
</table>
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>LDMIA</td>
<td>STMFD</td>
<td>STMDB</td>
</tr>
<tr>
<td>Empty ascending (EA)</td>
<td>LDMEA</td>
<td>LDMDB</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>

STMFD R13!, {R2-R9}
... @ modify R2-R9
LDMFD R13!, {R2-R9}
Branch and link

BL  sub1  @ call sub1
use stack to save/restore the return address and registers
...

sub1:  STMFD R13!, {R0-R2,R14}
BL  sub2
...
LDMFD R13!, {R0-R2,PC}

sub2:  ...
...
...
MOV  PC, LR
Conditional execution

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

  `movcs R0, R1`

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Condition</th>
<th>Mnemonic</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>Carry Set</td>
<td>CC</td>
<td>Carry Clear</td>
</tr>
<tr>
<td>EQ</td>
<td>Equal (Zero Set)</td>
<td>NE</td>
<td>Not Equal (Zero Clear)</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow Set</td>
<td>VC</td>
<td>Overflow Clear</td>
</tr>
<tr>
<td>GT</td>
<td>Greater Than</td>
<td>LT</td>
<td>Less Than</td>
</tr>
<tr>
<td>GE</td>
<td>Greater Than or Equal</td>
<td>LE</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td>PL</td>
<td>Plus (Positive)</td>
<td>MI</td>
<td>Minus (Negative)</td>
</tr>
<tr>
<td>HI</td>
<td>Higher Than</td>
<td>LO</td>
<td>Lower Than (aka CC)</td>
</tr>
<tr>
<td>HS</td>
<td>Higher or Same (aka CS)</td>
<td>LS</td>
<td>Lower or Same</td>
</tr>
</tbody>
</table>
Conditional execution

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

bypass: ...  
```

```
CMP   R0, #5
ADDNE R1, R1, R0
SUBNE R1, R1, R2
```

Rule of thumb: if the conditional sequence is three instructions or less, it is better to use conditional execution than a branch.
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 Mnemonic</th>
<th>Meaning</th>
<th>Operation Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add with Carry</td>
<td>MVN</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>RSB</td>
<td>Reverse Subtract</td>
</tr>
<tr>
<td>BAL</td>
<td>Unconditional Branch</td>
<td>RSC</td>
<td>Reverse Subtract with Carry</td>
</tr>
<tr>
<td>B⟨cc⟩</td>
<td>Branch on Condition</td>
<td>SBC</td>
<td>Subtract with Carry</td>
</tr>
<tr>
<td>BIC</td>
<td>Bit Clear</td>
<td>SMLAL</td>
<td>Mult Accum Signed Long</td>
</tr>
<tr>
<td>BLAL</td>
<td>Unconditional Branch and Link</td>
<td>SMULL</td>
<td>Multiply Signed Long</td>
</tr>
<tr>
<td>BL⟨cc⟩</td>
<td>Conditional Branch and Link</td>
<td>STM</td>
<td>Store Multiple</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>STR</td>
<td>Store Register (Word)</td>
</tr>
<tr>
<td>EOR</td>
<td>Exclusive OR</td>
<td>STRB</td>
<td>Store Register (Byte)</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 Unsigned Long</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>UMULL</td>
<td>Multiply Unsigned Long</td>
</tr>
</tbody>
</table>
ARM assembly program

<table>
<thead>
<tr>
<th>label</th>
<th>operation</th>
<th>operand</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:</td>
<td>LDR</td>
<td>R1, value</td>
<td>@ load value</td>
</tr>
<tr>
<td></td>
<td>STR</td>
<td>R1, result</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SWI</td>
<td>#11</td>
<td></td>
</tr>
</tbody>
</table>

value: .word 0x0000C123
result: .word 0
64-bit addition

```
ADR  R0, value1
LDR  R1, [R0]
LDR  R2, [R0, #4]
ADR  R0, value2
LDR  R3, [R0]
LDR  R4, [R0, #4]
ADDS R6, R2, R4
ADC  R5, R1, R3
STR  R5, [R0]
STR  R6, [R0, #4]
```

\[
\begin{array}{c}
01F0000000 \\
+ 0010000000 \\
\hline
0200000000 \\
\end{array}
\]

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

LDR   R1, value1
LDR   R2, value2
CMP   R1, R2
BHI   Done
MOV   R1, R2

Done:
STR   R1, result

value1: .word 4
value2: .word 9
result: .word 0
Sample

...  
while (i!=j)  
{  
    if (i>j)  
        i -= j;  
    else  
        j -= i;  
}  
...
Sample (Assembly)

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