Chapter 6
ARM Instruction Set

Hsung-Pin Chang
Department of Computer Science
National Chung Hsing University
Outline

- Data Processing Instructions
- Branch Instructions
- Load-store instructions
- Software interrupt instructions
- Program status register instructions
- Conditional Execution
# ARM Instruction Set Format

<table>
<thead>
<tr>
<th>Cond</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0 U</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0 B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 S</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1 L</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1 L</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1 L</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1 L</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>U</td>
<td>N</td>
<td>W L</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CP Opc</td>
<td>CRn</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CP Opc</td>
<td>Rd</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Data processing/FSR Transfer |
| Multiply |
| Multiply Long |
| Single data swap |
| Branch and exchange |
| Halfword data transfer: register offset |
| Halfword data transfer: immediate offset |
| Single data transfer |
| Undefined |
| Block data transfer |
| Branch |
| Coprocessor data transfer |
| Coprocessor Operation |
| Coprocessor register transfer |
| Software Interrupt |
6.1 Data Processing Instructions

- Manipulate data *within* registers
- Data processing instructions
  - Move instructions
  - Arithmetic instructions
  - Logical instructions
  - Comparison instructions
  - Multiply instructions
6.1.1 Move Instruction

- Syntax: `<instruction> {<cond>} {S} Rd, N`
  - N: a register or immediate value

- MOV: move
  - MOV r0, r1;  r0 = r1
  - MOV r0, #5;  r0 = 5

- MVN: move (negated)
  - MVN r0, r1;  r0 = NOT(r1) = ~ (r1)
Preprocessed by Shifter

- **Example 1**
  - PRE: \( r5 = 5, r7 = 8; \)
  
  - **MOV r7, r5, LSL #2;** \( r7 = r5 \ll 2 = r5 \times 4 \)
  
  - POST: \( r5 = 5, r7 = 20 \)
6.1.2 Preprocessed by Shifter

- **LSL**: logical shift left
  - \( x << y \), the least significant bits are filled with zeroes

- **LSR**: logical shift right:
  - (unsigned) \( x >> y \), the most significant bits are filled with zeroes

- **ASR**: arithmetic shift right
  - (signed) \( x >> y \), copy the sign bit to the most significant bit

- **ROR**: rotate right
  - ((unsigned) \( x >> y \)) | (\( x << (32-y) \))

- **RRX**: rotate right extended
  - c flag \( <<31 \) | ((unsigned) \( x >> 1 \))
  - Performs 33-bit rotate, with the CPSR’s C bit being inserted above sign bit of the word
Example 2

- PRE: \( r0 = 0x00000000, r1 = 0x80000004 \)

- `MOV r0, r1, LSL #1 ; r0 = r1 * 2`

- POST \( r0 = 0x00000008, r1 = 0x80000004 \)
6.1.3 Arithmetic Instructions

- **Syntax:** `<instruction> {<cond>} {S} Rd, Rn, N`
  - **N:** a register or immediate value

- **ADD:** add
  - `ADD r0, r1, r2; r0 = r1 + r2`

- **ADC:** add with carry
  - `ADC r0, r1, r2; r0 = r1 + r2 + C`

- **SUB:** subtract
  - `SUB r0, r1, r2; r0 = r1 - r2`

- **SBC:** subtract with carry
  - `SBC r0, r1, r2; r0 = r1 - r2 + C -1`
6.1.3 Arithmetic Instructions (Cont.)

- **RSB**: reverse subtract
  - \( RSB\ r0,\ r1,\ r2;\ r0 = r2 - r1 \)

- **RSC**: reverse subtract with carry
  - \( RSC\ r0,\ r1,\ r2;\ r0 = r2 - r1 + C\ -1 \)

- **MUL**: multiply
  - \( MUL\ r0,\ r1,\ r2;\ r0 = r1 \times r2 \)

- **MLA**: multiply and accumulate
  - \( MLA\ r0,\ r1,\ r2,\ r3;\ r0 = r1 \times r2 + r3 \)
6.1.4 Logical Operations

- Syntax: `<instruction> {<cond>} {S} Rd, RN, N`
  - N: a register or immediate value
- AND: Bit-wise and
- ORR: Bit-wise or
- EOR: Bit-wise exclusive-or
- BIC: bit clear
  - BIC r0, r1, r2; r0 = r1 & Not(r2)
Logical Operations (Cont)

- **Example 3:**
  - PRE: \( r1 = 0b1111, r2 = 0b0101 \)
  - BIC \( r0, r1, r2 \) ; \( r0 = r1 \ AND \ (NOT(r2)) \)
  - POST: \( r0 = 0b1010 \)
6.1.5 Comparison Instructions

- Compare or test a register with a 32-bit value
  - Do not modify the registers being compared or tested
  - But only set the values of the \textit{NZCV} bits of the \textit{CPSR register}
    - Do not need to apply to \textit{S} suffix for comparison instruction to update the flags in CPSR register
Comparison Instructions (Cont.)

- Syntax: `<instruction> {<cond>} {S} Rd, N
  - N: a register or immediate value
- CMP : compare
  - CMP r0, r1; compute (r0 - r1) and set NZCV
- CMN : negated compare
  - CMP r0, r1; compute (r0 + r1) and set NZCV
- TST : bit-wise AND test
  - TST r0, r1; compute (r0 AND r1) and set NZCV
- TEQ : bit-wise exclusive-or test
  - TEQ r0, r1; compute (r0 EOR r1) and set NZCV
Comparison Instructions (Cont.)

- **Example 4**
  - **PRE:** CPSR = nzcvqiFt_USER, r0 = 4, r9 = 4
  - **CMP** r0, r9
  - **POST:** CPSR = nZcvqiFt_USER
6.1.6 Multiply Instruction

- **Syntax:**
  - MLA{<cond>} {S} Rd, Rm, Rs, Rn
  - MUL{<cond>} {S} Rd, Rm, Rs

- **MUL** : multiply
  - MUL r0, r1, r2; \( r0 = r1 \times r2 \)

- **MLA** : multiply and accumulate
  - MLA r0, r1, r2, r3; \( r0 = (r1 \times r2) + r3 \)
Multiply Instruction (Cont.)

- **Syntax:** `<instruction>{<cond>} {S} RdLo, RdHi, Rm, Rs
  - Multiply onto a pair of register representing a 64-bit value

- **UMULL : unsigned multiply long**
  - `UMULL r0, r1, r2, r3; [r1,r0] = r2*r3`

- **UMLAL : unsigned multiply accumulate long**
  - `UMLAL r0, r1, r2, r3; [r1,r0] = [r1,r0]+(r2*r3)`

- **SMULL: signed multiply long**
  - `SMULL r0, r1, r2, r3; [r1,r0] = r2*r3`

- **SMLAL : signed multiply accumulate long**
  - `SMLAL r0, r1, r2, r3; [r1,r0] = [r1,r0]+(r2*r3)`
6.2 Branch Instructions

- Branch instruction
  - Change the flow of execution
  - Used to call a routine

- Allow applications to
  - Have subroutines
  - Implement if-then-else structure
  - Implement loop structure
Branch Instructions (Cont.)

- **Syntax**
  - `B{<cond>} label`
  - `BL{<cond>} label`

- **B : branch**
  - `B label; pc (program counter) = label`
  - Used to change execution flow

- **BL : branch and link**
  - `BL label; pc = label, lr = address of the next address after the BL`
  - Similar to the B instruction but can be used for subroutine call
    - Overwrite the link register (lr) with a return address
Example 5

B forward
ADD r1, r2, #4
ADD r0, r6, #2
ADD r3, r7, #4

Forward
SUB r1, r2, #4

Backward
SUB r1, r2, #4
B backward
Branch Instructions (Cont.)

- *Example 6:*

  `BL subroutine`
  `CMP r1, #5`
  `MOVEQ r1, #0`
  ...
  subroutine
  `<subroutine code>`
  `MOV pc, lr ; return by moving pc = lr`
6.3 Load-Store Instructions

- Transfer data between memory and processor registers

- Three types
  - Single-register transfer
  - Multiple-register transfer
  - Swap
6.3.1 Simple-Register Transfer

- Moving a single data item in and out of register

- Data item can be
  - A word (32-bits)
  - Halfword (16-bits)
  - Bytes (8-bits)
Simple-Register Transfer (Cont.)

- Syntax
  - \(<\text{LDR} \mid \text{STR}>\{<\text{cond}>\}{\{\text{B}\}} \text{ Rd, addressing}^1\)
  - \(\text{LDR}\{<\text{cond}>\}\text{SB} \mid \text{H} \mid \text{SH} \text{ Rd, addressing}^2\)
  - \(\text{STR}\{<\text{cond}>\} \text{ H Rd, addressing}^2\)

- LDR: load word into a register from memory
- LDRB: load byte
- LDRSB: load signed byte
- LDRH: load half-word
- LSRSH: load signed halfword

- STR: store word from a register to memory
- STRB: store byte
- STRH: store half-word
Example 7

LDR r0, [r1] := LDR r0, [r1, #0]
; r0 = mem32[r1]

STR r0, [r1] := STR r0, [r1, #0]
; mem32[r1] = r0

Register r1 is called the base address register
6.3.2 Single-Register Load-Store Addressing Mode

- **Index method**, also called **Base-Plus-Offset Addressing**
  - Base register
    - r0 – r15
  - Offset, add or subtract an unsigned number
    - Immediate
    - Register (not PC)
    - Scaled register
Single-Register Load-Store Addressing Mode (Cont.)

- **Preindex:**
  - data: \( \text{mem}[\text{base}+\text{offset}] \)
  - Base address register: *not updated*
  - Ex: \text{LDR} \ r0, [r1,#4] ; r0:=\text{mem32}[r1+4]

- **Postindex:**
  - data: \( \text{mem}[\text{base}] \)
  - Base address register: \( \text{base} + \text{offset} \)
  - Ex: \text{LDR} \ r0, [r1,#4] ; r0:=\text{mem32}[r1], \text{then} \ r1:=r1+4

- **Preindex with writeback** (also called *auto-indexing*)
  - Data: \( \text{mem}[\text{base}+\text{offset}] \)
  - Base address register: \( \text{base} + \text{offset} \)
  - Ex: \text{LDR} \ r0, [r1,#4]! ; r0:=\text{mem32}[r1+4], \text{then} \ r1:=r1+4
Example 8

- r0 = 0x00000000, r1 = 0x00009000, mem32[0x00009000] = 0x01010101, mem32[0x00009004] = 0x02020202

Preindexing: LDR r0, [r1, #4]
- r0 = 0x02020202, r1=0x00009000

Postindexing: LDR r0, [r1], #4
- r0 = 0x01010101, r1=0x00009004

Preindexing with writeback: LDR r0, [r1, #4]!
- R0 = 0x02020202, r1=0x00009004
## Single-Register Load-Store Addressing Mode (Cont.)

<table>
<thead>
<tr>
<th>Addressing mode and index method</th>
<th>Addressing syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preindex with <em>immediate offset</em></td>
<td>[Rn, #+/-offset_12]</td>
</tr>
<tr>
<td>Preindex with <em>register offset</em></td>
<td>[Rn, +/-Rm]</td>
</tr>
<tr>
<td>Preindex with <em>scaled register offset</em></td>
<td>[Rn, +/-Rm, shift #shift_imm]</td>
</tr>
<tr>
<td>Preindex writeback with immediate offset</td>
<td>[Rn, #+/-offset_12]!</td>
</tr>
<tr>
<td>Preindex writeback with register offset</td>
<td>[Rn, +/-Rm]!</td>
</tr>
<tr>
<td>Preindex writeback with <em>scaled register offset</em></td>
<td>[Rn, +/-Rm, shift #shift_imm]</td>
</tr>
<tr>
<td>Immediate postindexed</td>
<td>[Rn], #+/-offset_12]</td>
</tr>
<tr>
<td>Register postindexed</td>
<td>[Rn], +/-Rm!</td>
</tr>
<tr>
<td>Scaled register postindexed</td>
<td>[Rn], +/-Rm, shift #shift_imm</td>
</tr>
</tbody>
</table>
# Examples of LDR Using Different Addressing Modes

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>r0=</th>
<th>r1+=</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preindex with writeback</strong></td>
<td>LDR r0, [r1, #0x4]!</td>
<td>mem32[r1+0x4]</td>
<td>0x4</td>
</tr>
<tr>
<td></td>
<td>LDR r0, [r1, r2]!</td>
<td>mem32[r1+r2]</td>
<td>r2</td>
</tr>
<tr>
<td></td>
<td>LDR r0, [r1, r2, LSR#0x4]!</td>
<td>mem32[r1+(r2 LSR 0x4)]</td>
<td>(r2 LSR 0x4)</td>
</tr>
<tr>
<td><strong>Preindex</strong></td>
<td>LDR r0, [r1, #0x4]</td>
<td>mem32[r1+0x4]</td>
<td><em>not updated</em></td>
</tr>
<tr>
<td></td>
<td>LDR r0, [r1, r2]</td>
<td>mem32[r1+r2]</td>
<td><em>not updated</em></td>
</tr>
<tr>
<td></td>
<td>LDR r0, [r1, -r2, LSR #0x4]</td>
<td>Mem32[r1-(r2 LSR 0x4)]</td>
<td><em>not updated</em></td>
</tr>
<tr>
<td><strong>Postindex</strong></td>
<td>LDR r0, [r1], #0x4</td>
<td>mem32[r1]</td>
<td>0x4</td>
</tr>
<tr>
<td></td>
<td>LDR r0, [r1], r2</td>
<td>Mem32[r1]</td>
<td>r2</td>
</tr>
<tr>
<td></td>
<td>LDR r0, [r1], r2 LSR #0x4</td>
<td>mem32[r1]</td>
<td>(r2 LSR 0x4)</td>
</tr>
</tbody>
</table>
6.3.3 Multiple-Register Transfer

- Transfer multiple registers between memory and the processor in a single instruction

- More efficient than single-register transfer
  - Moving blocks of data around memory
  - Saving and restoring context and stack
Load-store multiple instruction can increase interrupt latency

- Interrupt can be occurred after an instruction has been completed
- Each load multiple instruction takes $2 + N \times t$ cycles
  - $N$: the number of registers to load
  - $t$: the number of cycles required for sequential access to memory

Compilers provides a switch to control the maximum number of registers between transferred
- Limit the maximum interrupt latency
Multiple-Register Transfer (Cont.)

- Syntax:
  - `<LDM | STM>{<cond>} <mode> Rn{!}, <registers>{^}
  - Address mode: See the next page
  - ^: optional
  - Can not be used in User Mode and System Mode
  - If op is LDM and reglist contains the pc (r15)
    - SPSR is also copied into the CPSR.
  - Otherwise, data is transferred into or out of the User mode registers instead of the current mode registers.
# Addressing Mode

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Description</th>
<th>Start address</th>
<th>End address</th>
<th>Rn!</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA</td>
<td>increment address after each transfer</td>
<td>$Rn$</td>
<td>$Rn + 4N - 4$</td>
<td>$Rn + 4N$</td>
</tr>
<tr>
<td>IB</td>
<td>increment address before each transfer</td>
<td>$Rn + 4$</td>
<td>$Rn + 4N$</td>
<td>$Rn + 4N$</td>
</tr>
<tr>
<td>DA</td>
<td>decrement address after each transfer</td>
<td>$Rn - 4N + 4$</td>
<td>$Rn$</td>
<td>$Rn - 4N$</td>
</tr>
<tr>
<td>DB</td>
<td>decrement address before each transfer</td>
<td>$Rn - 4N$</td>
<td>$Rn - 4$</td>
<td>$Rn + 4N$</td>
</tr>
</tbody>
</table>
Multiple-Register Transfer (Cont.)

- **Example 9**
  - **PRE:**
    - mem32[0x80018] = 0x03,
    - mem32[0x80014] = 0x02,
    - mem32[0x80010] = 0x01,
    - r0 = 0x00080010,
    - r1 = r2 = r3 = 0x00000000
  
  - **LDMIA r0!, {r1-r3}, or LDMIA r0!, {r1, r2, r3}**
    - Register can be explicitly listed or use the “-” character
## Pre-Condition for LDMIA Instruction

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80020</td>
<td>0x00000005</td>
</tr>
<tr>
<td>0x8001c</td>
<td>0x00000004</td>
</tr>
<tr>
<td>0x80018</td>
<td>0x00000003</td>
</tr>
<tr>
<td>0x80014</td>
<td>0x00000002</td>
</tr>
<tr>
<td>0x80010</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0x8000c</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

R0 = 0x80010

R1 = 0x00000000
R2 = 0x00000000
R3 = 0x00000000

*Figure 1*
Post-Condition for LDMIA Instruction

$R0 = 0x8001c$

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80020</td>
<td>0x00000005</td>
</tr>
<tr>
<td>0x8001c</td>
<td>0x00000004</td>
</tr>
<tr>
<td>0x80018</td>
<td>0x00000003</td>
</tr>
<tr>
<td>0x80014</td>
<td>0x00000002</td>
</tr>
<tr>
<td>0x80010</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0x8000c</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

$R3 = 0x00000003$
$R2 = 0x00000002$
$R1 = 0x00000001$

Figure 2
Multiple-Register Transfer (Cont.)

- Example 9 (Cont.)
  - POST:
    - $r_0 = 0x0008001c,$
    - $r_1 = 0x00000001,$
    - $r_2 = 0x00000002,$
    - $r_3 = 0x00000003$
Example 10

- **PRE:** as shown in Fig. 1
- **LDMIB r0!, {r1-r3}**
- **POST:**
  
  - \( r0 = 0x0008001c \)
  - \( r1 = 0x00000004 \)
  - \( r2 = 0x00000003 \)
  - \( r3 = 0x00000002 \)
Post-Condition for LDMIB Instruction

\[ R0 = 0x8001c \]

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Data</th>
<th>R1 = 0x000000002</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80020</td>
<td>0x00000005</td>
<td></td>
</tr>
<tr>
<td>0x8001c</td>
<td>0x00000004</td>
<td></td>
</tr>
<tr>
<td>0x80018</td>
<td>0x00000003</td>
<td></td>
</tr>
<tr>
<td>0x80014</td>
<td>0x00000002</td>
<td></td>
</tr>
<tr>
<td>0x80010</td>
<td>0x00000001</td>
<td></td>
</tr>
<tr>
<td>0x8000c</td>
<td>0x00000000</td>
<td></td>
</tr>
</tbody>
</table>

\[ R2 = 0x00000003 \]

\[ R3 = 0x00000004 \]

Figure 3
Multiple-Register Transfer (Cont.)

- Load-store multiple pairs when base update used (!)
  - Useful for saving a group of registers and store them later

<table>
<thead>
<tr>
<th>Store multiple</th>
<th>Load multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>STMIA</td>
<td>LDMDB</td>
</tr>
<tr>
<td>STMIB</td>
<td>LDMDA</td>
</tr>
<tr>
<td>STMDA</td>
<td>LDMIB</td>
</tr>
<tr>
<td>STMDB</td>
<td>LDMIA</td>
</tr>
</tbody>
</table>
Example 11

PRE:
\[
\begin{align*}
    r0 &= 0x00009000 \\
    r1 &= 0x00000009, \\
    r2 &= 0x00000008 \\
    r3 &= 0x00000007
\end{align*}
\]

STMIB r0!, {r1-r3}
MOV   r1, #1
MOV   r2, #2,
MOV   r3, #3
Example 11 (Cont.)

PRE (2):
- r0 = 0x0000900c
- r1 = 0x00000001
- r2 = 0x00000002
- r3 = 0x00000003

LDMDA r0!, {r1-r3}

POST:
- r0 = 0x00009000
- r1 = 0x00000009
- r2 = 0x00000008
- r3 = 0x00000007
Example 11 (Cont.)

- The STMIB stores the values 7, 8, 9 to memory

- Then corrupt register $r1$ to $r3$ by MOV instruction

- Finally, the LDMDA
  - Reloads the original values, and
  - Restore the base pointer $r0$
Example 12: the use of the load-store multiple instructions with a block memory copy

- $r9$ points to start of source data
- $r10$ points to start of destination data
- $r11$ points to end of the source

```assembly
  LDMIA  r9!, {r0-r7}    ;load 32 bytes from source and update $r9$
  STMIA  r10!, {r0-r7}  ;store 32 bytes to desti. and update $r10$
  CMP    r9, r11        ;have we reached the end
  BNE    loop
```

loop
Multiple-Register Transfer (Cont.)

Copy memory Location
(transfer 32 bytes in two instructions)
6.3.4 Stack Operations

- ARM architecture uses the *load-store multiple instruction* to carry out *stack operations*
  - *PUSH*: use a *store multiple* instruction
  - *POP*: use a *load multiple* instruction

- Stack
  - *Ascending (A)*: stack grows towards higher memory addresses
  - *Descending (D)*: stack grows towards lower memory addresses
6.3.4 Stack Operations (Cont.)

- **Stack**
  - *Full stack* (F): stack pointer \( sp \) points to the last valid item pushed onto the stack
  - *Empty stack* (E): \( sp \) points *after* the last item on the stack
    - The free slot where the next data item will be placed

- There are a number of aliases available to support stack operations
  - See next page
ARM supports all four forms of stacks:

- **Full ascending (FA):** grows up; base register points to the highest address containing a valid item.
- **Empty ascending (EA):** grows up; base register points to the first empty location.
- **Full descending (FD):** grows down; base register points to the lowest address containing a valid data.
- **Empty descending (ED):** grows down; base register points to the first empty location below the stack.
# Addressing Methods for Stack Operations

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Description</th>
<th>Pop</th>
<th>=LDM</th>
<th>Push</th>
<th>=STM</th>
</tr>
</thead>
<tbody>
<tr>
<td>FA</td>
<td>Full ascending</td>
<td>LDMFA</td>
<td>LDMDA</td>
<td>STMFA</td>
<td>STMIB</td>
</tr>
<tr>
<td>FD</td>
<td>Full descending</td>
<td>LDMFD</td>
<td>LDMIA</td>
<td>STMFD</td>
<td>STMDB</td>
</tr>
<tr>
<td>EA</td>
<td>Empty ascending</td>
<td>LDMEA</td>
<td>LDMDB</td>
<td>STMEA</td>
<td>STMIA</td>
</tr>
<tr>
<td>ED</td>
<td>Empty descending</td>
<td>LDMED</td>
<td>LDMIB</td>
<td>STMED</td>
<td>STMDA</td>
</tr>
</tbody>
</table>
6.3.4 Stack Operations (Cont.)

- **Example 13**
  - **PRE:**
    - $r1 = 0x00000002$
    - $r4 = 0x00000003$
    - $sp = 0x00080014$
  - **STMFD** $sp!, \{r1, r4\}$
  - **POST:**
    - $r1 = 0x00000002$
    - $r4 = 0x00000003$
    - $sp = 0x00080014c$
6.3.4 Stack Operations (Cont.)

- **Example 13 (Cont.)**
  - STMFD – full stack push operation

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80018</td>
<td>0x00000001</td>
<td>0x80018</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0x80014</td>
<td>0x00000002</td>
<td>0x80014</td>
<td>0x00000002</td>
</tr>
<tr>
<td>0x80010</td>
<td>Empty</td>
<td>0x80010</td>
<td>0x00000003</td>
</tr>
<tr>
<td>0x8000c</td>
<td>Empty</td>
<td>0x8000c</td>
<td>0x00000002</td>
</tr>
</tbody>
</table>
6.3.4 Stack Operations (Cont.)

Example 14

PRE:
- \( r1 = 0x00000002 \)
- \( r4 = 0x00000003 \)
- \( sp = 0x00080010 \)

STMED \( sp! \), \{r1, r4\}

POST:
- \( r1 = 0x00000002 \)
- \( r4 = 0x00000003 \)
- \( sp = 0x00080008 \)
6.3.4 Stack Operations (Cont.)

- **Example 14 (Cont.)**
  - STMED – empty stack push operation

<table>
<thead>
<tr>
<th>PRE</th>
<th>POST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Address</strong></td>
<td><strong>Address</strong></td>
</tr>
<tr>
<td>0x80018</td>
<td>0x80018</td>
</tr>
<tr>
<td>0x80014</td>
<td>0x80014</td>
</tr>
<tr>
<td>0x80010</td>
<td>0x80010</td>
</tr>
<tr>
<td>0x8000c</td>
<td>0x8000c</td>
</tr>
<tr>
<td>0x80008</td>
<td>0x80008</td>
</tr>
<tr>
<td><strong>Data</strong></td>
<td><strong>Data</strong></td>
</tr>
<tr>
<td>0x00000001</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0x00000002</td>
<td>0x00000002</td>
</tr>
<tr>
<td>Empty</td>
<td>Empty</td>
</tr>
<tr>
<td>Empty</td>
<td>Empty</td>
</tr>
<tr>
<td>Empty</td>
<td>Empty</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sp</th>
<th>sp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000c</td>
<td>0x80008</td>
</tr>
</tbody>
</table>
6.3.3 SWAP Instruction

- A special case of a load-store instruction
  - Swap the contents of memory with the contents of a register
  - An atomic operation
    - Cannot not be interrupted by any other instruction or any other buy access
    - The system “holds the bus” until the transaction is complete
    - Useful when implementing semaphores and mutual exclusion in an operating system
6.3.3 SWAP Instruction (Cont.)

- Syntax: \textbf{SWP}\{B}\{<\text{cond}>\} \text{Rd, Rm, [Rn]}
  - \( tmp = \text{mem32}[Rn] \)
  - \( \text{Mem32}[Rn] = Rm \)
  - \( Rd = tmp \)

- \textbf{SWP}: swap a word between memory and a register

- \textbf{SWPB}: swap a byte between memory and a register
6.3.3 SWAP Instruction (Cont.)

- **Example 15**
  - **PRE:**
    - Mem32[0x9000] = 0x12345678
    - r0 = 0x00000000
    - r1 = 0x11112222
    - r2 = 0x00009000
  - **SWP r0, r1, [r2]**
  - **POST:**
    - mem32[0x9000] = 0x11112222
    - r0 = 0x12345678
    - r1 = 0x11112222
    - r2 = 0x00009000
6.3.3 SWAP Instruction (Cont.)

- **Example 15 (Cont.)**
  
  ```assembly
  SPIN
  MOV  r1, =semaphore
  MOV  r2, #1
  SWP  r3, r2, [r1] ;hold the bus until complete
  CMP  r3, #1
  BEQ  spin
  ```
  
- The address pointed by the semaphore either contains the value of 1 or 0

- When semaphore value == 1, loop until semaphore becomes 0 (updated by the holding process)
6.4 Software Interrupt Instruction

- SWI: software interrupt instruction
  - Cause a software interrupt exception
  - Provide a mechanism for applications to call operating system routines
  - Each SWI instruction has an associated SWI number
    - Used to represent a particular function call or routines
6.4 Software Interrupt Instruction (Cont.)

- **Syntax:** \texttt{SWI\{<cond>\} SWI\_number}
  - \texttt{lr\_svc} = address of instruction following the SWI
  - \texttt{spsr\_svc} = \texttt{cpsr}
  - \texttt{pc} = vector table + 0x8 \quad ; \textit{jump to the swi handling}
  - \texttt{cpsr mode} = SVC
  - \texttt{cpsr I} = 1 (mask IRQ interrupt)
Example 16

**PRE:**
- cpsr = nzcVqift_USER
- pc = 0x00008000
- lr = r14 = 0x003fffff

**POST:**
- cpsr = nzcVqift_SVC
- spsr = nzcVqift_USER
- pc = 0x00000008
- lr = 0x00008004
6.5 Program Status Register Instructions

- **MRS**
  - Transfer the contents of either the *cpsr* or *spsr* into a *register*

- **MSR**
  - Transfer the contents of a *register* into the *cpsr* or *spsr*
6.5 Program Status Register Instructions (Cont.)

- **Syntax**
  - MRS{<cond>} Rd, <cpsr | spsr>
  - MSR{<cond>} <cpsr | spsr>_<fields>, Rm
  - MSR{<cond>} <cpsr | spsr>_<fields>, #immediate

- **Field**: any combination of
  - Flags: [24:31]
  - Status: [16:23]
  - eXtension[8:15]
  - Control[0:7]
PSR Registers

- Condition Code Flags
  - N (Negative)
  - Z (Zero)
  - C (Carry/Borrow/Extend)
  - V (Overflow)
  - Negative/Less Than

- (Reserved)

- Control Bits
  - I (IRQ disable)
  - F (FIQ disable)
  - T (Mode bits)
  - M4
  - M3
  - M2
  - M1
  - M0

State bit
Note: You cannot access the SPSR in User or System Mode

Assembler cannot warn you because it does not know which mode will be executed in
6.5 Program Status Register Instructions (Cont.)

- **Example 17**
  - **PRE:**
    - cpsr = nzcvqIFt_SVC
    - MRS r1, cpsr
    - BIC r1, r1, #0x80 ;0b10000000, clear bit 7
    - MSR cpsr_c, r1 ;enable IRQ interrupts
  - **POST:**
    - cpsr = nzcvqIFt_SVC
  - Note that, this example must be in SVC mode
  - In user mode, you can only read all cpsr bits and can only update the condition flag field, i.e., cpsr[24:31]
6.6 Conditional Execution

- Almost all ARM instruction can include an optional condition code
  - Instruction is only executed if the condition code flags in the CPSR meet the specified condition
  - The default is **AL**, or *always execute*

- Conditional executions depends on two components
  - The *condition field*: located in the instruction
  - The *condition flags*: located in the *cpsr*
Conditional Execution (Cont.)

Example 18

ADDEQ  r0, r1, r2
; r0 = r1 + r2 if zero flag is set
# Condition Codes

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Flags</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Z set</td>
<td>Equal</td>
</tr>
<tr>
<td>NE</td>
<td>Z clear</td>
<td>Not equal</td>
</tr>
<tr>
<td>CS/HS</td>
<td>C set</td>
<td>Higher or same (unsigned &gt;=)</td>
</tr>
<tr>
<td>CC/L0</td>
<td>C clear</td>
<td>Lower (unsigned &lt;)</td>
</tr>
<tr>
<td>MI</td>
<td>N set</td>
<td>Negative</td>
</tr>
<tr>
<td>PL</td>
<td>N clear</td>
<td>Positive or zero</td>
</tr>
<tr>
<td>VS</td>
<td>V set</td>
<td>Overflow</td>
</tr>
<tr>
<td>VC</td>
<td>V clear</td>
<td>No overflow</td>
</tr>
<tr>
<td>HI</td>
<td>C set and Z clear</td>
<td>Higher (unsigned &lt;=)</td>
</tr>
<tr>
<td>LS</td>
<td>C clear or Z set</td>
<td>Lower or same (unsigned &lt;=)</td>
</tr>
<tr>
<td>GE</td>
<td>N and V the same</td>
<td>Signed &gt;=</td>
</tr>
<tr>
<td>LT</td>
<td>N and V different</td>
<td>Signed &lt;</td>
</tr>
<tr>
<td>GT</td>
<td>Z clear, and N and V the same</td>
<td>Signed &gt;</td>
</tr>
<tr>
<td>LE</td>
<td>Z set, or N and V different</td>
<td>Signed &lt;=</td>
</tr>
<tr>
<td>AL</td>
<td>Any</td>
<td>Always (usually omitted)</td>
</tr>
</tbody>
</table>
6.6 Conditional Execution (Cont.)

- Thus, before activate conditional execution
  - There must be an instruction that updates the conditional code flag according to the result
  - If not specified, instructions will not update the flags

- To make an instruction update the flags
  - Include the S suffix
  - Example: ADDS r0, r1,r2
6.6 Conditional Execution (Cont.)

- However, some instructions always update the flags:
  - Do not require the S suffix
  - CMP, CMN, TST, TEQ

- Flags are preserved until updated

- Thus, you can execute an instruction conditionally, based upon the flags set in another instruction, either:
  - Immediately after the instruction which updated the flags
  - After any number of intervening instructions that have not updated the flags.
6.6 Conditional Execution (Cont.)

- **Example 18**
  - Transfer the following code into the assembly language
  - Assume \( r1 = a, r2 = b \)

```c
while ( a != b )
{
    if (a > b) a -= b; else b -= a;
}
```
6.6 Conditional Execution (Cont.)

- **Example 18: Solution 1**

```
gcd
  CMP r1, r2
  BEQ complete
  BLT lessthan
  SUB r1, r1, r2
  B gcd
lessthan
  SUB r2, r1, r2
  B gcd
complete
```
6.6 Conditional Execution (Cont.)

- **Example 18: Solution 2**

  \[
  \begin{align*}
  & \text{gcd} \\
  & \text{CMP} \quad r1, r2 \\
  & \text{SUBGT} \quad r1, r1, r2 \\
  & \text{SUBLT} \quad r2, r2, r1 \\
  & \text{BNE} \quad \text{gcd}
  \end{align*}
  \]

- Solution 2 dramatically reduces the number of instructions !!!
References

  - Chapter 3: Introduction to the ARM Instruction Set