Introduction

- The ARM processor is 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 simulator.

ARM Instruction Set

Computer Organization and Assembly Languages
Yung-Yu Chuang

with slides by Peng-Sheng Chen

Introduction

- The ARM processor is 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 simulator.

ARM Instruction Set

Computer Organization and Assembly Languages
Yung-Yu Chuang

with slides by Peng-Sheng Chen

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</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>0xFFFFFFF0</td>
<td>00</td>
</tr>
<tr>
<td>0xFFFFF00</td>
<td>00</td>
</tr>
<tr>
<td>0xFFFFFFFF</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

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 registers at once
- Possible to combine shift and ALU operations in a single instruction
### Instruction set

- Data processing
- Data movement (memory access)
- Flow control

### Data processing

- They are move, arithmetic, logical, comparison and multiply instructions.
- Most data processing instructions can process one of their operands using the barrel shifter.

#### 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

### Instruction set

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

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

**MOV S R0, #0 @ R0:=0
   @ Z=1, N=0
   @ C, V unaffected**

### 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>HLS</td>
<td>Higher or Same (aka CS)</td>
<td>LS</td>
<td>Lower or Same</td>
</tr>
</tbody>
</table>
Register movement

Syntax: `<instruction>{<cond>}{<reg>]<Rd, N immediate, register, shift>

<table>
<thead>
<tr>
<th>MOV</th>
<th>Move a 32-bit value into a register</th>
<th>Rd = N</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVN</td>
<td>move the NOT of the 32-bit value into a register</td>
<td>Rd = ~N</td>
</tr>
</tbody>
</table>

• MOV R0, R2 @ R0 = R2
• MVN R0, R2 @ R0 = ~R2

Immediate operands

ADD R0, R1, R2

Immediate operands

a literal; most can be represented by (0..255)x2^n 0<n<12

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

a hexadecimal literal
This is assembler dependent syntax.

Addressing modes

• Register operands
  ADD R0, R1, R2

• Immediate operands

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 fast multiplication and dealing with lists, table and other complex data structure. (similar to the displacement addressing mode in CISC.)

Some instructions (e.g. MUL, CLZ, QADD) do not read barrel shifter.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Shift</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL</td>
<td>logical shift left</td>
<td>xLSL y</td>
<td>x &lt;&lt; y</td>
</tr>
<tr>
<td>LSR</td>
<td>logical shift right</td>
<td>xLSR y</td>
<td>(unsigned)x &gt;&gt; y</td>
</tr>
<tr>
<td>ASR</td>
<td>arithmetic right shift</td>
<td>xASR y</td>
<td>(signed)x &gt;&gt; y</td>
</tr>
<tr>
<td>ROR</td>
<td>rotate right</td>
<td>xROR y</td>
<td>((unsigned)x &gt;&gt; y)</td>
</tr>
<tr>
<td>RXX</td>
<td>rotate right extended</td>
<td>xRXX</td>
<td>(c flag &lt;&lt; 31)</td>
</tr>
</tbody>
</table>
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=0xA0000031
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

- 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.

@ array index calculation
ADD R0, R1, R2, LSL R3 @ R0 := R1 + R2 \times 2^{R3}

@ fast multiply R2 = 35xR0
ADD R0, R0, R0, LSL #2 @ R0' = 5xR0
RSB R2, R0, R0, LSL #3 @ R2 = 7xR0'
Multiplication

MOV R1, #35
MUL R2, R0, R1

or
ADD R0, R0, R0, LSL #2 @ R0' = 5xR0
RSB R2, R0, R0, LSL #3 @ R2 = 7xR0'

Shifted register operands

<table>
<thead>
<tr>
<th>N shift operations</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>#immediate</td>
</tr>
<tr>
<td>Register</td>
<td>Rm</td>
</tr>
<tr>
<td>Logical shift left by immediate</td>
<td>Rm, LSL #shift_imm</td>
</tr>
<tr>
<td>Logical shift left by register</td>
<td>Rm, LSL Rs</td>
</tr>
<tr>
<td>Logical shift right by immediate</td>
<td>Rm, LSR #shift_imm</td>
</tr>
<tr>
<td>Logical shift right with register</td>
<td>Rm, LSR Rs</td>
</tr>
<tr>
<td>Arithmetic shift right by immediate</td>
<td>Rm, ASR #shift_imm</td>
</tr>
<tr>
<td>Arithmetic shift right by register</td>
<td>Rm, ASR Rs</td>
</tr>
<tr>
<td>Rotate right by immediate</td>
<td>Rm, ROR #shift_imm</td>
</tr>
<tr>
<td>Rotate right by register</td>
<td>Rm, ROR Rs</td>
</tr>
<tr>
<td>Rotate right with extend</td>
<td>Rm, RRX</td>
</tr>
</tbody>
</table>

Encoding data processing instructions

Arithmetic

- Add and subtraction

Syntax: <instruction><cond>[S] Rd, Rn, N

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>add two 32-bit values and carry</td>
<td>Rd = Rn + N + carry</td>
</tr>
<tr>
<td>ADD</td>
<td>add two 32-bit values</td>
<td>Rd = Rn + N</td>
</tr>
<tr>
<td>RSB</td>
<td>reverse subtract of two 32-bit values</td>
<td>Rd = N − Rn</td>
</tr>
<tr>
<td>RSC</td>
<td>reverse subtract with carry of two 32-bit values</td>
<td>Rd = N − Rn − !\text{(carry flag)}</td>
</tr>
<tr>
<td>SBC</td>
<td>subtract with carry of two 32-bit values</td>
<td>Rd = Rn − N − !\text{(carry flag)}</td>
</tr>
<tr>
<td>SUB</td>
<td>subtract two 32-bit values</td>
<td>Rd = Rn − N</td>
</tr>
</tbody>
</table>
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
- **RSB** R0, R1, R2 @ R0 = R2-R1
- **RSC** R0, R1, R2 @ R0 = R2-R1-!C

### 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

### Example

**PRE**
- r0 = 0x00000000
- r1 = 0x00000001

**POST**
- r0 = 0x0000000f
- r1 = 0x00000005

**SUBS r1, r1, #1**

**POST**
- r0 = 0x00000000
- r1 = 0x00000005

**ADD r0, r1, r1, LSL #1**

**POST**
- r0 = 0x0000000f
- r1 = 0x00000005

**RSB r0, r1, #0 ; Rd = 0x0 - r1**

**POST**
- r0 = -r1 = 0xffffffff89
Logical

Syntax: `<instruction> [<cond>] {S} Rd, Rn, N

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND Rd, R1, R2</td>
<td>logical bitwise AND of two 32-bit values</td>
<td>Rd = R1 &amp; R2</td>
</tr>
<tr>
<td>ORR Rd, R1, R2</td>
<td>logical bitwise OR of two 32-bit values</td>
<td>Rd = R1</td>
</tr>
<tr>
<td>EOR Rd, R1, R2</td>
<td>logical exclusive OR of two 32-bit values</td>
<td>Rd = R1 ^ R2</td>
</tr>
<tr>
<td>BIC Rd, R1, R2</td>
<td>logical bit clear (AND NOT)</td>
<td>Rd = R1 &amp; ~R2</td>
</tr>
</tbody>
</table>

Logical 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.

Comparison Syntax: `<instruction> [<cond>] Rn, N

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN</td>
<td>compare negated</td>
<td>flags set as a result of <code>Rn + N</code></td>
</tr>
<tr>
<td>CMP</td>
<td>compare</td>
<td>flags set as a result of <code>Rn - N</code></td>
</tr>
<tr>
<td>TEQ</td>
<td>test for equality of two 32-bit values</td>
<td>flags set as a result of <code>Rn ^ N</code></td>
</tr>
<tr>
<td>TST</td>
<td>test bits of a 32-bit value</td>
<td>flags set as a result of <code>Rn &amp; N</code></td>
</tr>
</tbody>
</table>

Example:

**Logical**

- 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

BIC R0, R1, R2

R0=0x10011010
Comparison

- **CMP**  R1, R2  @ set cc on R1−R2
- **CMN**  R1, R2  @ set cc on R1+R2
- **TST**  R1, R2  @ set cc on R1 and R2
- **TEQ**  R1, R2  @ set cc on R1 xor R2

Multiplication

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

<table>
<thead>
<tr>
<th>MLA</th>
<th>multiply and accumulate</th>
<th>Rd = (Rm*Rs) + Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>multiply</td>
<td>Rd = Rm*Rs</td>
</tr>
</tbody>
</table>

Syntax:  <instruction>{<cond>}{S}  RdLo, RdHi, Rm, Rs

<table>
<thead>
<tr>
<th>SMLAL</th>
<th>signed multiply accumulate long</th>
<th>[RdHi, RdLo] = [RdHi, RdLo] + (Rm*Rs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMULL</td>
<td>signed multiply long</td>
<td>[RdHi, RdLo] = Rm*Rs</td>
</tr>
<tr>
<td>UMLAL</td>
<td>unsigned multiply accumulate long</td>
<td>[RdHi, RdLo] − [RdHi, RdLo] + (Rm*Rs)</td>
</tr>
<tr>
<td>UMULL</td>
<td>unsigned multiply long</td>
<td>[RdHi, RdLo] = Rm*Rs</td>
</tr>
</tbody>
</table>

Comparison

PRE  cpsr = nzcvqiFt_USER
    r0 = 4
    r9 = 4

CMP  r0, r9

POST  cpsr = nZcvqiFt_USER

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
- Cycles depends on core type
- If S bit is set, C flag is meaningless

See the reference manual (4.1.33)
**Multiplication**

- Multiply-accumulate (2D array indexing)
  
  \[
  MLA \ R4, R3, R2, R1 @ R4 = R3 \times R2 + R1
  \]

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

  \[
  MOV \ R1, \#35
  MUL \ R2, R0, R1
  \]
  
or
  
  \[
  ADD \ R0, R0, R0, LSL \#2 \ @ \ R0' = 5 \times R0
  RSB \ R2, R0, R0, LSL \#3 \ @ \ R2 = 7 \times R0'
  \]

**Multiplication**

**Flow control instructions**

- Determine the instruction to be executed next

**Syntax:**

- `B{<cond>} label`
- `BL{<cond>} label`
- `BX{<cond>} Rm`
- `BLX{<cond>} label | Rm`

<table>
<thead>
<tr>
<th>B</th>
<th>branch</th>
<th>pc=label</th>
<th>pc-relative offset within 32MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL</td>
<td>branch with link</td>
<td>pc=label</td>
<td>lr=address of the next instruction after the BL</td>
</tr>
<tr>
<td>BX</td>
<td>branch exchange</td>
<td>pc=Rm &amp; 0xffffffff, T=Rm &amp; 1</td>
<td></td>
</tr>
<tr>
<td>BLX</td>
<td>branch exchange with link</td>
<td>pc=label, T=1</td>
<td>µ=¬Rm &amp; 0xffffffff, T¬Rm &amp; 1</td>
</tr>
<tr>
<td></td>
<td>lr=address of the next instruction after the BLX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Determine the instruction to be executed next

**Syntax:**

- `B{<cond>} label`
- `BL{<cond>} label`
- `BX{<cond>} Rm`
- `BLX{<cond>} label | Rm`

<table>
<thead>
<tr>
<th>B</th>
<th>branch</th>
<th>pc=label</th>
<th>pc-relative offset within 32MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL</td>
<td>branch with link</td>
<td>pc=label</td>
<td>lr=address of the next instruction after the BL</td>
</tr>
<tr>
<td>BX</td>
<td>branch exchange</td>
<td>pc=Rm &amp; 0xffffffff, T=Rm &amp; 1</td>
<td></td>
</tr>
<tr>
<td>BLX</td>
<td>branch exchange with link</td>
<td>pc=label, T=1</td>
<td>µ=¬Rm &amp; 0xffffffff, T¬Rm &amp; 1</td>
</tr>
<tr>
<td></td>
<td>lr=address of the next instruction after the BLX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Flow control instructions

- Branch instruction
  \[ B \text{ label} \]
  \[ \text{label: } \ldots \]

- Conditional branches
  \[ \text{MOV R0, #0} \]
  \[ \text{loop: } \ldots \]
  \[ \text{ADD R0, R0, #1} \]
  \[ \text{CMP R0, #10} \]
  \[ \text{BNE loop} \]

Branch conditions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Condition flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>equal</td>
<td>Z</td>
</tr>
<tr>
<td>NE</td>
<td>not equal</td>
<td>z</td>
</tr>
<tr>
<td>CS, HS</td>
<td>carry set/unsigned higher or same</td>
<td>C</td>
</tr>
<tr>
<td>CC, LO</td>
<td>carry clear/unsigned lower</td>
<td>c</td>
</tr>
<tr>
<td>MI</td>
<td>minus/negative</td>
<td>N</td>
</tr>
<tr>
<td>PL</td>
<td>plus/positive or zero</td>
<td>n</td>
</tr>
<tr>
<td>VS</td>
<td>overflow</td>
<td>V</td>
</tr>
<tr>
<td>VC</td>
<td>no overflow</td>
<td>y</td>
</tr>
<tr>
<td>HI</td>
<td>unsigned higher</td>
<td>zC</td>
</tr>
<tr>
<td>LS</td>
<td>unsigned lower or same</td>
<td>Z or c</td>
</tr>
<tr>
<td>GE</td>
<td>signed greater than or equal</td>
<td>NV or nV</td>
</tr>
<tr>
<td>LT</td>
<td>signed less than</td>
<td>NV or nV</td>
</tr>
<tr>
<td>GT</td>
<td>signed greater than</td>
<td>NZ or nZV</td>
</tr>
<tr>
<td>LE</td>
<td>signed less than or equal</td>
<td>Z or NZ or nZV</td>
</tr>
<tr>
<td>AL</td>
<td>always (unconditional)</td>
<td>ignored</td>
</tr>
</tbody>
</table>

Branch and link

- BL instruction saves the return address to R14 (lr)
  \[ \text{BL sub } @ \text{ call sub} \]
  \[ \text{CMP R1, #5 } @ \text{ return to here} \]
  \[ \text{MOVEQ R1, #0} \]
  \[ \ldots \]
  \[ \text{sub: } \ldots @ \text{ sub entry point} \]
  \[ \ldots \]
  \[ \text{MOV PC, LR} @ \text{ return} \]
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

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

cmp    R0, #5    smaller and faster
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
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

Syntax: `<LDR|STR>{<cond>}(B) Rd, addressing`

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>Load word into a register</td>
</tr>
<tr>
<td>STR</td>
<td>Save byte or word from a register</td>
</tr>
<tr>
<td>LDRB</td>
<td>Load byte into a register</td>
</tr>
<tr>
<td>STRB</td>
<td>Save byte from a register</td>
</tr>
</tbody>
</table>

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

Single register load/store

The data items can be a 8-bit byte, 16-bit half-word or 32-bit word. Addresses must be boundary aligned. (e.g. 4's multiple for LDR/STR)

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

No STRSB/STRSH since STRB/STRH stores both signed/unsigned ones
Addressing modes

- Memory is addressed by a register and an offset.
  \[
  \text{LDR } R0, [R1] \rightarrow \text{mem}[R1]
  \]
- Three ways to specify offsets:
  - Immediate
    \[
    \text{LDR } R0, [R1, #4] \rightarrow \text{mem}[R1+4]
    \]
  - Register
    \[
    \text{LDR } R0, [R1, R2] \rightarrow \text{mem}[R1+R2]
    \]
  - Scaled register
    \[
    \text{LDR } R0, [R1, R2, LSL #2] \rightarrow \text{mem}[R1+4*R2]
    \]

Pre-index addressing

\[
\text{LDR } R0, [R1, #4] \rightarrow R0=\text{mem}[R1+4]
\]
@ R1 unchanged

Auto-indexing addressing

\[
\text{LDR } R0, [R1, #4]! \rightarrow R0=\text{mem}[R1+4]
\]
@ R1=R1+4
No extra time; Fast;

<table>
<thead>
<tr>
<th>Index method</th>
<th>Data</th>
<th>Base address register</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preindex with writeback</td>
<td>mem[base + offset]</td>
<td>base + offset</td>
<td>LDR r0,[r1,#4]!</td>
</tr>
<tr>
<td>Preindex</td>
<td>mem[base + offset]</td>
<td>not updated</td>
<td>LDR r0,[r1,#4]</td>
</tr>
<tr>
<td>Postindex</td>
<td>mem[base]</td>
<td>base + offset</td>
<td>LDR r0,[r1,#4]</td>
</tr>
</tbody>
</table>
Post-index addressing

\[
\text{LDR } R0, R1, #4 \quad \@ \quad R0 = \text{mem}[R1] \\
\quad \@ \quad R1 = R1 + 4 \\
\text{LDR } R0, [R1], \\
\text{R0} \\
\]

Comparisons

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

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

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

Example

| PRE | r0 = 0x00000000  \\
r1 = 0x00090000  \\
mem32[0x00009000] = 0x01010101  \\
mem32[0x00009004] = 0x02020202 |
|-----|---------------------------------|
| POST(1) | r0 = 0x02020202  \\
r1 = 0x00090004 |

Preindexing with writeback:

| POST(2) | r0 = 0x02020202  \\
r1 = 0x00090000 |
Example

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

LDR r0, [r1], #4

Postindexing:

POST(3) r0 = 0x01010101
r1 = 0x00009004
Summary of addressing modes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result</th>
<th>r1 +=</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preindex with writeback</td>
<td>STRH r0,[r1,#0x4]!</td>
<td>mem16[r1+0x4] = r0</td>
</tr>
<tr>
<td>POSTINDEX</td>
<td>STRH r0,[r1,r2]!</td>
<td>mem16[r1+r2] = r0</td>
</tr>
<tr>
<td>STRH r0,[r1,#0x4]</td>
<td>mem16[r1+0x4] = r0</td>
<td>not updated</td>
</tr>
<tr>
<td>STRH r0,[r1,r2]</td>
<td>mem16[r1+r2] = r0</td>
<td>not updated</td>
</tr>
</tbody>
</table>

Load an address into a register

- Note that all addressing modes are register-offseted. Can we issue `LDR R0, Table`? The pseudo instruction `ADR` loads a register with an address:

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

- Assembler transfer pseudo instruction into a sequence of appropriate instructions:

  ```
  sub r0, pc, #12
  ```

Application

```plaintext
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 a block of data more efficiently.
- Used for procedure entry and exit for saving and restoring workspace registers and the return address.
- For ARM7, \(2+Nt\) cycles (\(N\):#words, \(t\):time for a word for sequential access). Increase interrupt latency since it can’t be interrupted.

  ```
  LDMIA R1, {R0, R2, R5} @ R0 = mem[R1]
  @ R2 = mem[r1+4]
  @ R5 = mem[r1+8]
  ```

  ```plaintext
  registers are arranged an in increasing order; see manual
  ```
**Multiple load/store register**

LDM   load multiple registers
STM   store multiple registers

<table>
<thead>
<tr>
<th>suffix</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA</td>
<td>increase after</td>
</tr>
<tr>
<td>IB</td>
<td>increase before</td>
</tr>
<tr>
<td>DA</td>
<td>decrease after</td>
</tr>
<tr>
<td>DB</td>
<td>decrease before</td>
</tr>
</tbody>
</table>

**Addressing modes**

Syntax: `<LDM|STM>{cond}<addressing mode> Rn[!,]<registers>{^}`

<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 after</td>
<td>Rn</td>
<td>Rn + 4*N - 4</td>
<td>Rn + 4*N</td>
</tr>
<tr>
<td>IB</td>
<td>increment before</td>
<td>Rn + 4</td>
<td>Rn + 4*N</td>
<td>Rn + 4*N</td>
</tr>
<tr>
<td>DA</td>
<td>decrement after</td>
<td>Rn - 4*N + 4</td>
<td>Rn</td>
<td>Rn - 4*N</td>
</tr>
<tr>
<td>DB</td>
<td>decrement before</td>
<td>Rn - 4*N</td>
<td>Rn - 4</td>
<td>Rn - 4*N</td>
</tr>
</tbody>
</table>

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

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
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

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

Multiple load/store register

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

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

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

Multiple load/store register

LDMIA R0!, {R1,R2,R3}

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

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

LDMIB R0!, {R1,R2,R3}

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

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

Example

PRE
mem32[0x80018] = 0x03
mem32[0x80014] = 0x02
mem32[0x80010] = 0x01
r0 = 0x00000010
r1 = 0x00000000
r2 = 0x00000000
r3 = 0x00000000

LDMIA r0!, {r1-r3}

...
Example

<table>
<thead>
<tr>
<th>Address pointer</th>
<th>Memory address</th>
<th>Data</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>

LDMIA  r0!, {r1-r3}

Example

<table>
<thead>
<tr>
<th>Address pointer</th>
<th>Memory address</th>
<th>Data</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>

LDMIB  r0!, {r1-r3}

Application

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

loop: LDMIA R9!, {R0-R7}
  STMIA R10!, {R0-R7}
  CMP   R9, R11
  BNE   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</td>
<td>LDMFA</td>
<td>LDMDA</td>
<td>STMFA</td>
<td>STMIB</td>
</tr>
<tr>
<td>Full descending</td>
<td>LDMFD</td>
<td>LDMIA</td>
<td>STMFD</td>
<td>STMDB</td>
</tr>
<tr>
<td>Empty ascending</td>
<td>LDMEA</td>
<td>LDMDB</td>
<td>STMEA</td>
<td>STMIA</td>
</tr>
<tr>
<td>Empty descending</td>
<td>LDMEI</td>
<td>LDMDI</td>
<td>STMED</td>
<td>STMDA</td>
</tr>
</tbody>
</table>

STMFD R13!, {R2-R9} @ used for ATPCS
   ... @ modify R2-R9
LDMFD R13!, {R2-R9}
### Example

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

STMFD sp!, {r1,r4}

<table>
<thead>
<tr>
<th>POST Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80018</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0x80014</td>
<td>0x00000002</td>
</tr>
<tr>
<td>0x80010</td>
<td>0x00000003</td>
</tr>
<tr>
<td>sp</td>
<td>0x8000c</td>
</tr>
</tbody>
</table>

### Swap instruction

- Swap between memory and register. Atomic operation preventing any other instruction from reading/writing to that location until it completes

Syntax: `SWP[B]{<cond>} Rd, Rn, [Rn]`

<table>
<thead>
<tr>
<th>SWP</th>
<th>Description</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWP</td>
<td>swap a word between memory and a register</td>
<td><code> swap tmp = mem32[Rn] mem32[Rn] = Rd Rd = tmp</code></td>
</tr>
<tr>
<td>SWPB</td>
<td>swap a byte between memory and a register</td>
<td><code> swap tmp = mem8[Rn] mem8[Rn] = Rd Rd = tmp</code></td>
</tr>
</tbody>
</table>

### Application

```
spin

MOV r1, =semaphore
MOV r2, #1
SWP r3, r2, [r1] ; hold the bus until complete
CMP r3, #1
BEQ spin

while (1) {
  if (s==0) {
    s=1;
    // use the resource
  }
  // use the resource
}
```
Software interrupt

- A software interrupt instruction causes a software interrupt exception, which provides a mechanism for applications to call OS routines.

Syntax: SWI(<cond>) SWI_number

<table>
<thead>
<tr>
<th>SWI</th>
<th>software interrupt</th>
<th>h_svc = address of instruction following the SWI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>spsr_svc = cpsr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pc = vectors + 0x8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cpsr mode = SVC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cpsr 1 = 1 (mask IRQ interrupts)</td>
<td></td>
</tr>
</tbody>
</table>

Example

PRE
- cpsr = nzCqi.ft_USER
- pc = 0x00000000
- lr = 0x003fffffff; lr = r14
- r0 = 0x12

0x00000000 SWI 0x123456

POST
- cpsr = nzCqi.ft_SVC
- spsr = nzCqi.ft_USER
- pc = 0x00000008
- lr = 0x00000004
- r0 = 0x12

Load constants

- No ARM instruction loads a 32-bit constant into a register because ARM instructions are 32-bit long. There is a pseudo code for this.

Syntax: LDR Rd, =constant
- ADR Rd, label

<table>
<thead>
<tr>
<th>LDR</th>
<th>load constant pseudoinstruction</th>
<th>Rd = 32-bit constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADR</td>
<td>load address pseudoinstruction</td>
<td>Rd = 32-bit relative address</td>
</tr>
</tbody>
</table>

Immediate numbers

- v = n ror 2r

encoding for data processing instructions
Load constants

- Assemblers implement this usually with two options depending on the number you try to load.

<table>
<thead>
<tr>
<th>Pseudoinstruction</th>
<th>Actual instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR r0, =0xff</td>
<td>MOV r0, #0xff</td>
</tr>
<tr>
<td>LDR r0, =0x55555555</td>
<td>LDR r0, [pc, #offset_12]</td>
</tr>
</tbody>
</table>

Loading the constant 0xfff000000

- Assume that you want to load 511 into R0

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov r0, #256</td>
<td>add r0, #255</td>
</tr>
<tr>
<td>ldr r0, L511 .word 511</td>
<td>ldr r0, [pc, #0]</td>
</tr>
</tbody>
</table>

- Guideline: if you can construct it in two instructions, do it; otherwise, load it.
- The assembler decides for you

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldr r0, =255 mov r0, 255</td>
<td>ldr r0, =511 ldr r0, [pc, #4]</td>
</tr>
</tbody>
</table>

PC-relative modes

- Pre-indexed addressing mode (P=1)
- Post-indexed addressing mode (P=2)
- Impossible to use direct addressing encoding for data transfer instructions

PC-relative addressing

```
main:
    MOV R0, #0
    ADR R1, a          @ add r1, pc, #4
    STR R0, [R1]
    PC SWI #11
a: .word 100
.end
```
# 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>NEQ</td>
<td>Logical NOT</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>ORA</td>
<td>Logical OR</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
<td>RSB</td>
<td>Reverse Subtract</td>
</tr>
<tr>
<td>BR</td>
<td>Unconditional Branch</td>
<td>RSC</td>
<td>Reverse Subtract with Carry</td>
</tr>
<tr>
<td>B(ce)</td>
<td>Branch on Condition</td>
<td>SBC</td>
<td>Subtract with Carry</td>
</tr>
<tr>
<td>BIC</td>
<td>Bit Clear</td>
<td>SMULL</td>
<td>Multiply Signed Long</td>
</tr>
<tr>
<td>BLAL</td>
<td>Unconditional Branch and Link</td>
<td>STM</td>
<td>Store Multiple</td>
</tr>
<tr>
<td>BL(ce)</td>
<td>Conditional Branch and Link</td>
<td>SVI</td>
<td>Software Interrupt</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>STLB</td>
<td>Store Register (Byte)</td>
</tr>
<tr>
<td>LDM</td>
<td>Load Multiple</td>
<td>SUBB</td>
<td>Subtract</td>
</tr>
<tr>
<td>Load Register (Word)</td>
<td>Load Register (Byte)</td>
<td>SWI</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>TEO</td>
<td>Test Equivalence</td>
</tr>
<tr>
<td>MSR</td>
<td>Load SPR or CPSR</td>
<td>TEST</td>
<td>Test</td>
</tr>
<tr>
<td>MOV</td>
<td>Store to SPR or CPSR</td>
<td>UMULL</td>
<td>Multiply Signed Long</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>UMULL</td>
<td>Multiply Unsigned Long</td>
</tr>
</tbody>
</table>