Introduction to the ARM Instruction Set

• The complete list of ARM instructions available in the ARMv5E instruction set architecture (ISA) is provided in the following table
• This ISA includes all the core ARM instructions as well as some additional features in the ARM instruction set
• The “ARM ISA” column lists the ISA revision in which the revision is introduced
• Some instructions have extended functionality in later architectures
# Introduction to the ARM Instruction Set

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>ARM ISA</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>v1</td>
<td>add two 32-bit values and carry</td>
</tr>
<tr>
<td>ADD</td>
<td>v1</td>
<td>add two 32-bit values</td>
</tr>
<tr>
<td>AND</td>
<td>v1</td>
<td>logical bitwise AND of two 32-bit values</td>
</tr>
<tr>
<td>B</td>
<td>v1</td>
<td>branch relative +/- 32 MB</td>
</tr>
<tr>
<td>BIC</td>
<td>v1</td>
<td>logical bit clear (AND NOT) of two 32-bit values</td>
</tr>
<tr>
<td>BKPT</td>
<td>v5</td>
<td>breakpoint instructions</td>
</tr>
<tr>
<td>BL</td>
<td>v1</td>
<td>relative branch with link</td>
</tr>
<tr>
<td>BLX</td>
<td>v5</td>
<td>branch with link and exchange</td>
</tr>
<tr>
<td>BX</td>
<td>v4T</td>
<td>branch with exchange</td>
</tr>
<tr>
<td>CDP</td>
<td>v2 v5</td>
<td>coprocessor data processing operation</td>
</tr>
<tr>
<td>CDP2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLZ</td>
<td>v5</td>
<td>count leading zeros</td>
</tr>
<tr>
<td>CMN</td>
<td>v1</td>
<td>compare negative two 32-bit values</td>
</tr>
<tr>
<td>CMP</td>
<td>v1</td>
<td>compare two 32-bit values</td>
</tr>
<tr>
<td>EOR</td>
<td>v1</td>
<td>logical exclusive OR of two 32-bit values</td>
</tr>
<tr>
<td>LDC</td>
<td>v2 v5</td>
<td>load to coprocessor single or multiple 32-bit values</td>
</tr>
<tr>
<td>LDC2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDM</td>
<td>v1</td>
<td>load multiple 32-bit words from memory to ARM registers</td>
</tr>
<tr>
<td>LDR</td>
<td>v1 v4 v5E</td>
<td>load a single value from a virtual address in memory</td>
</tr>
<tr>
<td>MCR</td>
<td>v2 v5 v5E</td>
<td>move to coprocessor from an ARM register or registers</td>
</tr>
<tr>
<td>MCR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCRR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MLA</td>
<td>v2</td>
<td>multiply and accumulate 32-bit values</td>
</tr>
<tr>
<td>MOV</td>
<td>v1</td>
<td>move a 32-bit value into a register</td>
</tr>
</tbody>
</table>
**Introduction to the ARM Instruction Set**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRC v2 v5 v5E</td>
<td>move to ARM register or registers from a coprocessor</td>
</tr>
<tr>
<td>MRS v3</td>
<td>move to ARM register from a status register (cpsr or spsr)</td>
</tr>
<tr>
<td>MSR v3</td>
<td>move to a status register (cpsr or spsr) from an ARM register</td>
</tr>
<tr>
<td>MUL v2</td>
<td>multiply two 32-bit values</td>
</tr>
<tr>
<td>MVN v1</td>
<td>move the logical NOT of 32-bit value into a register</td>
</tr>
<tr>
<td>ORR v1</td>
<td>logical bitwise OR of two 32-bit values</td>
</tr>
<tr>
<td>PLD v5E</td>
<td>preload hint instruction</td>
</tr>
<tr>
<td>QADD v5E</td>
<td>signed saturated 32-bit add</td>
</tr>
<tr>
<td>QDADD v5E</td>
<td>signed saturated double and 32-bit add</td>
</tr>
<tr>
<td>QDSUB v5E</td>
<td>signed saturated double and 32-bit subtract</td>
</tr>
<tr>
<td>QSUB v5E</td>
<td>signed saturated 32-bit subtract</td>
</tr>
<tr>
<td>RSB v1</td>
<td>reverse subtract of two 32-bit values</td>
</tr>
<tr>
<td>RSC v1</td>
<td>reverse subtract with carry of two 32-bit integers</td>
</tr>
<tr>
<td>SBC v1</td>
<td>subtract with carry of two 32-bit values</td>
</tr>
<tr>
<td>SMLAxy v5E</td>
<td>signed multiply accumulate instructions ((16 × 16) + 32 = 32-bit)</td>
</tr>
<tr>
<td>SMLAL v3M</td>
<td>signed multiply accumulate long ((32 × 32) + 64 = 64-bit)</td>
</tr>
<tr>
<td>SMLALxy v5E</td>
<td>signed multiply accumulate long ((16 × 16) + 64 = 64-bit)</td>
</tr>
<tr>
<td>SMLAWy v5E</td>
<td>signed multiply accumulate instruction (((32 × 16) → 16) + 32 = 32-bit)</td>
</tr>
<tr>
<td>SMULL v3M</td>
<td>signed multiply long (32 × 32 = 64-bit)</td>
</tr>
</tbody>
</table>
## Introduction to the ARM Instruction Set

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>ARM ISA</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMULxy</td>
<td>v5E</td>
<td>signed multiply instructions (16 × 16 = 32-bit)</td>
</tr>
<tr>
<td>SMULWxy</td>
<td>v5E</td>
<td>signed multiply instruction ((32 × 16) ≫ 16 = 32-bit)</td>
</tr>
<tr>
<td>STC STC2</td>
<td>v2 v5</td>
<td>store to memory single or multiple 32-bit values from coprocessor</td>
</tr>
<tr>
<td>STM</td>
<td>v1</td>
<td>store multiple 32-bit registers to memory</td>
</tr>
<tr>
<td>STR</td>
<td>v1 v4 v5E</td>
<td>store register to a virtual address in memory</td>
</tr>
<tr>
<td>SUB</td>
<td>v1</td>
<td>subtract two 32-bit values</td>
</tr>
<tr>
<td>SWI</td>
<td>v1</td>
<td>software interrupt</td>
</tr>
<tr>
<td>SWP</td>
<td>v2a</td>
<td>swap a word/byte in memory with a register, without interruption</td>
</tr>
<tr>
<td>TEQ</td>
<td>v1</td>
<td>test for equality of two 32-bit values</td>
</tr>
<tr>
<td>TST</td>
<td>v1</td>
<td>test for bits in a 32-bit value</td>
</tr>
<tr>
<td>UMLAL</td>
<td>v3M</td>
<td>unsigned multiply accumulate long ((32 × 32) + 64 = 64-bit)</td>
</tr>
<tr>
<td>UMULL</td>
<td>v3M</td>
<td>unsigned multiply long (32 × 32 = 64-bit)</td>
</tr>
</tbody>
</table>
Introduction to the ARM Instruction Set

• The processor operations will be illustrated using example with pre- and post-conditions → describing registers and memory before and after the instruction or instructions are executed

• The hexadecimal numbers is represented with the prefix 0x and binary number with the prefix 0b

• The following example shows the format

```
PRE <pre-conditions>
<instruction/s>
POST <post-conditions>
```
Introduction to the ARM Instruction Set

• In the pre- and post-conditions, memory is denoted as

\[
\text{mem<data\_size>}[\text{address}]
\]

• This refers to \text{data\_size} memory starting at the given byte address

• For example, \text{mem32}[1024] is the 32-bit value starting at address 1 KB

• ARM instructions process data held in registers and only access memory with load and store instructions

• ARM instructions commonly take two or three operands
Introduction to the ARM Instruction Set

- For example, the `ADD r3, r1, r2`
- The `ADD` instruction adds the two values stored in register `r1` and `r2` (the source register)
- It writes the result to register `r3` (the destination register)

<table>
<thead>
<tr>
<th>Instruction Syntax</th>
<th>Destination register (Rd)</th>
<th>Source register 1 (Rn)</th>
<th>Source register 2 (Rm)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ADD r3, r1, r2</code></td>
<td><code>r3</code></td>
<td><code>r1</code></td>
<td><code>r2</code></td>
</tr>
</tbody>
</table>
Data Processing Instructions

- The data processing instructions manipulate data within registers.
- They consist of:
  - move instructions
  - arithmetic instructions
  - logic instructions
  - comparison instructions
  - multiply instructions
- Most data processing instructions can process one of their operands using the barrel shifter.
Data Processing Instructions

• If the S suffix is used on a data processing instruction → it updates the flags in the cpsr
• Move and logical operations update the carry flag $C$, negative flag $N$, and zero flag $Z$
• The carry flag is set from the result of the barrel shift as the last bit shifted out
• The negative flag is set to bit 31 of the result
• The zero flag is set if the result is zero
Move Instructions

- Move is the simplest ARM instruction.
- It copies N into a destination register Rd, N can be a register or immediate value.
- This instruction is useful for setting initial values and transferring data between registers.

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>Move a 32-bit value into a register</td>
<td>$Rd = N$</td>
</tr>
<tr>
<td>MVN</td>
<td>move the NOT of the 32-bit value into a register</td>
<td>$Rd = \sim N$</td>
</tr>
</tbody>
</table>
Move Instructions

• Example: MOV r7, r5

<table>
<thead>
<tr>
<th>PRE</th>
<th>r5 = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r7 = 8</td>
</tr>
<tr>
<td></td>
<td>MOV r7, r5 ; let r7 = r5</td>
</tr>
<tr>
<td>POST</td>
<td>r5 = 5</td>
</tr>
<tr>
<td></td>
<td>r7 = 5</td>
</tr>
</tbody>
</table>
Barrel Shifter

- The N in `MOV` instruction can be more than just a register or immediate value
- N can also be a register Rm, that has been preprocessed by the barrel shifter before being used by `MOV` instruction
- Data processing instructions are processed within the ALU
- A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU
- This shifts increases the power and flexibility of many data processing operations
- There are data processing instructions that do not use the barrel shift as well, e.g. `MUL` (multiply), `CLZ` (count leading zero), and `QADD` (signed saturated 32-bit add)
Barrel Shifter

- Pre-processing or shift occurs within the cycle time of the instruction
- This is useful for loading constants into a register and then achieving fast multiplies or division by the power of 2
- Register Rn enters the ALU without any pre-processing of the register
Barrel Shifter

- Example: we apply a logic shift left (LSL) to register Rm before moving it to the destination register
- This is the same as applying the standard C language shift operator << to the register
- The MOV instruction copies the shift operator result N into register Rd
- N represents the result of the LSL operation

```
PRE  r5 = 5
     r7 = 8

MOV    r7, r5, LSL #2 ; let r7 = r5*4 = (r5<<2)

POST r5 = 5
      r7 = 20
```
Barrel Shifter

- The five different shift operations which can be used within the barrel shifter are summarized as follows:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Shift</th>
<th>Result</th>
<th>Shift amount $y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL</td>
<td>logical shift left</td>
<td>$x\text{LSL}_y$</td>
<td>$x \ll y$</td>
<td>#0–31 or $Rs$</td>
</tr>
<tr>
<td>LSR</td>
<td>logical shift right</td>
<td>$x\text{LSR}_y$</td>
<td>(unsigned)$x \gg y$</td>
<td>#1–32 or $Rs$</td>
</tr>
<tr>
<td>ASR</td>
<td>arithmetic right shift</td>
<td>$x\text{ASR}_y$</td>
<td>(signed)$x \gg y$</td>
<td>#1–32 or $Rs$</td>
</tr>
<tr>
<td>ROR</td>
<td>rotate right</td>
<td>$x\text{ROR}_y$</td>
<td>((unsigned)$x \gg y)$</td>
<td>($x \ll (32 - y)$)</td>
</tr>
<tr>
<td>RRX</td>
<td>rotate right extended</td>
<td>$x\text{RRX}$</td>
<td>$(c \text{ flag } \ll 31)</td>
<td>((\text{unsigned})x \gg 1)$</td>
</tr>
</tbody>
</table>

Note: $x$ represents the register being shifted and $y$ represents the shift amount.
Barrel Shifter

- The following figure shows a logical shift left by one
- The content of bit 0 are shifted to bit 1, bit 0 is clear
- The C flag is updated with the last bit shifted out of the register
Barrel Shifter

- Barrel shift operation syntax for data processing instructions

<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>
Barrel Shifter

- Example: `MOV r0, r1, LSL #1`
- This example shifts register `r1` left by one bit (multiply register `r1` by a value $2^1$)
- The C flag is updated in the `cpsr` because the S suffix is present in the instruction mnemonic

\[
\begin{align*}
\text{PRE} & \quad \text{cpsr} = \text{nzcvqif USER} \\
& \quad r0 = 0x00000000 \\
& \quad r1 = 0x80000004 \\
\text{MOV} & \quad r0, r1, LSL \ #1 \\
\text{POST} & \quad \text{cpsr} = \text{nzCvqiFt USER} \\
& \quad r0 = 0x00000008 \\
& \quad r1 = 0x80000004
\end{align*}
\]
Arithmetic Instructions

- The arithmetic instructions implement addition and subtraction of 32-bit signed and unsigned values.

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>add two 32-bit values and carry</td>
<td>$Rd = Rn + N + \text{carry}$</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 Instructions

• Example: \texttt{SUB r0, r1, r2}
• This simple subtract instruction subtracts a value stored in register \textit{r2} from a value stored in register \textit{r1}
• The result is stored in register \textit{r0}

\begin{tabular}{l}
\textbf{PRE} & \texttt{r0} = 0x00000000 \\
& \texttt{r1} = 0x00000002 \\
& \texttt{r2} = 0x00000001 \\
\hline
\textbf{SUB r0, r1, r2} & \\
\textbf{POST} & \texttt{r0} = 0x00000001
\end{tabular}
Arithmetic Instructions

• Example: \texttt{RSB r0, r1, #0}

• The reverse subtract instruction (\texttt{RSB}) subtracts \texttt{r1} from a constant value \texttt{#0}

• The result is stored in register \texttt{r0}

• This function can be used to negate numbers

\begin{verbatim}
PRE  r0 = 0x00000000
  r1 = 0x00000077

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

POST r0 = -r1 = 0xfffffffff89
\end{verbatim}
Arithmetic Instructions

- Example: `SUBS r1, r1, #1`
- The `SUBS` instruction is useful for decrementing loop counters.
- In this example we subtract the immediate value one from the value one stored in the register `r1`.
- The result value zero is written to register `r1`.
- The `cpsr` is updated with the ZC flags being set.

**PRE**

- `cpsr = nzcvqiFt_USER`
- `r1 = 0x00000001`

**SUBS r1, r1, #1**

**POST**

- `cpsr = nZCvqiFt_USER`
- `r1 = 0x00000000`
Using the Barrel Shifter with Arithmetic Instructions

- The wide range of second operand shifts available on arithmetic and logical instructions is a very powerful feature of the ARM instruction set.
- Example: `ADD r0, r1, r1, LSL #1`
- Register `r1` is first shifted one location to the left to give the value of twice `r1`.
- The `ADD` instruction then adds the result of the barrel shift operation to register `r1`.
- The final result transferred into register `r0` is equal to three times the value stored in register `r1`.

**PRE**
- `r0 = 0x00000000`
- `r1 = 0x00000005`

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

**POST**
- `r0 = 0x0000000f`
- `r1 = 0x00000005`
Logical Instructions

- Logical instructions perform bitwise logical operations on the two source registers

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>logical bitwise AND of two 32-bit values</td>
<td>$Rd = Rn &amp; N$</td>
</tr>
<tr>
<td>ORR</td>
<td>logical bitwise OR of two 32-bit values</td>
<td>$Rd = Rn \mid N$</td>
</tr>
<tr>
<td>EOR</td>
<td>logical exclusive OR of two 32-bit values</td>
<td>$Rd = Rn ^ N$</td>
</tr>
<tr>
<td>BIC</td>
<td>logical bit clear (AND NOT)</td>
<td>$Rd = Rn &amp; \sim N$</td>
</tr>
</tbody>
</table>
Logical Instructions

• Example: ORR r0, r1, r2
• This example shows a logical OR operation between register r1 and r2
• The result is stored in r0

<table>
<thead>
<tr>
<th>PRE</th>
<th>r0 = 0x00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r1 = 0x02040608</td>
</tr>
<tr>
<td></td>
<td>r2 = 0x10305070</td>
</tr>
<tr>
<td></td>
<td>ORR r0, r1, r2</td>
</tr>
</tbody>
</table>

| POST | r0 = 0x12345678 |
Logical Instructions

• Example: \texttt{BIC} r0, r1, r2
• This example shows a logical \texttt{BIC} operation which carries out a logical bit clear

\begin{align*}
\text{PRE} & \quad r1 = \text{Ob111} \\
& \quad r2 = \text{Ob0101} \\
\text{BIC} & \quad r0, r1, r2 \\
\text{POST} & \quad r0 = \text{Ob1010}
\end{align*}

• This is equivalent to

\[ R_d = R_n \text{ AND NOT(N)} \]
Logical Instructions

- Register \( r2 \) contains a binary pattern where every binary 1 in \( r2 \) clears a corresponding bit location in register \( r1 \).
- This is useful when clearing status bits and is frequently used to change interrupt mask in the \( cpsr \).
- The logical instructions update the \( cpsr \) flags only if the \( s \) suffix is present.
- These instructions can use barrel-shifted second operands in the same way as the arithmetic instructions.
Comparison Instructions

- The comparison instructions are used to compare or test a register with a 32-bit value.
- They update the *cpsr* flag bits according to the result, but do not affect other registers.
- After the bits have been set, the information can then be used to change program flow by using conditional execution.
- You do NOT need to apply the *s* suffix for comparison instructions to update the flags.
Comparison Instructions

- N is the result of the shifter operation
- The syntax of shifter operation is shown below

**Syntax:** `<instruction>{<cond>}` *Rn, N*

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

• Example: \texttt{CMP r0, r9}

• This example shows comparison instruction

• Register \texttt{r0} and \texttt{r9} are equal before executing the instructions

• The value of \texttt{z} flag prior to execution is 0 (lower case)

• After execution the \texttt{z} flag changes to 1 (upper case) \rightarrow indicating equality

\begin{verbatim}
PRE  cpsr = nzcvqiFt_USER  
r0 = 4
r9 = 4

CMP   r0, r9

POST  cpsr = nZcvqiFt_USER
\end{verbatim}
Comparison Instructions

• The `cmp` is effectively a subtract instruction with the result discarded
• Similarly the `tst` instruction is a logical AND operation and `teq` is a logical exclusive OR operation
• All the results are discarded but the condition bits are updated in the `cpsr`
• The comparison thus only modify the condition flags of the `cpsr` and do NOT affect the register being compared
Multiply Instructions

- The multiply instructions multiply the contents of a pair of registers and, depending upon the instruction, accumulate the results in with another register.
- The long multipliers accumulate onto a pair of registers representing a 64-bit value.
- The final result is placed in a destination register or a pair of registers.

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation Description</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLA</td>
<td>multiply and accumulate</td>
<td>Rd = (Rm*Rs) + Rn</td>
</tr>
<tr>
<td>MUL</td>
<td>multiply</td>
<td>Rd = Rm*Rs</td>
</tr>
</tbody>
</table>
Multiply Instructions

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMLAL</td>
<td>signed multiply accumulate long</td>
<td>([RdHi, RdLo] = [RdHi, RdLo] + (Rm*Rs))</td>
</tr>
<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>
Multiply Instructions

- Example: `MUL r0, r1, r2`
- This example shows multiplication of register \( r1 \) and \( r2 \)
- The result is placed into register \( r0 \)

<table>
<thead>
<tr>
<th>PRE</th>
<th>( r0 = 0x00000000 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( r1 = 0x00000002 )</td>
</tr>
<tr>
<td></td>
<td>( r2 = 0x00000002 )</td>
</tr>
<tr>
<td>MUL</td>
<td>( r0, r1, r2 ) ; ( r0 = r1 \times r2 )</td>
</tr>
<tr>
<td>POST</td>
<td>( r0 = 0x00000004 )</td>
</tr>
<tr>
<td></td>
<td>( r1 = 0x00000002 )</td>
</tr>
<tr>
<td></td>
<td>( r2 = 0x00000002 )</td>
</tr>
</tbody>
</table>
Multiply Instructions

- The long multiply instruction produce a 64-bit result.
- The result is too large to fit a single 32-bit register; so the result is placed in two register labeled RdLo and RdHi.
- RdLo holds the lower 32 bits of the 64-bit result, and RdHi holds the higher 32-bit of the 64-bit result.
Multiply Instructions

• Example: `UMULL r0, r1, r2, r3`
• This example multiplies register $r2$ and $r3$ and places the result into register $r0$ and $r1$
• Register $r0$ contains lower 32 bits, and register $r1$ contains the higher 32 bits of the 64-bit result

```
PRE
r0 = 0x00000000
r1 = 0x00000000
r2 = 0xf0000002
r3 = 0x00000002

UMULL r0, r1, r2, r3 ; [r1,r0] = r2*r3

POST
r0 = 0xe00000004 ; = RdLo
r1 = 0x000000001 ; = RdHi
```
Branch Instructions

- A branch instruction changes the flow of execution or is used to call a routine.
- This type of instruction allows programs to have subroutines, *if-then-else* structures and loops.
- The change of execution flow forces the program counter $pc$ to point to a new address.
- The address label is stored in the instruction as a signed $pc$-relative offset → must be within approximately 32 MB of branch instruction.
- $T$ refers to the thumb bit in the $cpsr$.
- When instructions set $T$, the ARM switches to Thumb state.
Branch Instructions

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>
</tr>
</thead>
<tbody>
<tr>
<td>BL</td>
<td>branch with link</td>
<td>pc = label</td>
</tr>
<tr>
<td>BX</td>
<td>branch exchange</td>
<td>pc = Rm &amp; 0xffffffffe, T = Rm &amp; 1</td>
</tr>
<tr>
<td>BLX</td>
<td>branch exchange with link</td>
<td>pc = label, T = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pc = Rm &amp; 0xffffffffe, T = Rm &amp; 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lr = address of the next instruction after the BLX</td>
</tr>
</tbody>
</table>
Branch Instructions

- Example: `B label`
- This example shows a forward and backward branch
- Because these loops are address specific → we do not include pre- and post-condition
- The forward branch skips three instructions
- The backward branch creates an infinite loop
- Most assemblers hide the detail of a branch instruction encoding by using labels (forward and backward in this example)
Branch Instructions

- Example: `BL label`

- The branch with link is similar to the `B` instruction but overwrites the link register `lr` with a return address.

- It performs subroutine call and when return from subroutine the content of the link register is copied to the `pc`.

```assembly
BL subroutine ; branch to subroutine
CMP r1, #5 ; compare r1 with 5
MOVEQ r1, #0 ; if (r1==5) then r1 = 0

: subroutine
  <subroutine code>
MOV pc, lr ; return by moving pc = lr
```
Branch Instructions

• The branch exchange (BX) uses an absolute address stored in register Rm → it is used to branch to or from Thumb code
• The $T$ bit is the $cpsr$ is updated by the least significant bit of the branch register
• Similarly branch exchange with link (BLX) instruction updates the $T$ bit of the $cpsr$ with the least significant bit and additionally sets the link register with the return address
Load-Store Instructions/ Single-Register Transfer

- Load-store instructions transfer data between memory and processor register.
- There are three types of load-store instructions:
  1. Single-register transfer
  2. Multiple-register transfer
  3. Swap
- Single–register transfer is used for moving a single data item in and out of a register.
- The data types supported are signed and unsigned words (32-bit), halfwords (16-bit) and bytes.
## Single-Register Transfer

Syntax: $\langle LDR | STR \rangle \{ <\text{cond}> \} \{ \langle B \rangle \} \text{Rd, addressing}^1$

- $\langle LDR \rangle \{ <\text{cond}> \} \text{SB|H|SH Rd, addressing}^2$
- $\langle STR \rangle \{ <\text{cond}> \} \text{H Rd, addressing}^2$

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>load word into a register</td>
<td>$\text{Rd} &lt;- \text{mem32}[\text{address}]$</td>
</tr>
<tr>
<td>STR</td>
<td>save byte or word from a register</td>
<td>$\text{Rd} \rightarrow \text{mem32}[\text{address}]$</td>
</tr>
<tr>
<td>LDRB</td>
<td>load byte into a register</td>
<td>$\text{Rd} &lt;- \text{mem8}[\text{address}]$</td>
</tr>
<tr>
<td>STRB</td>
<td>save byte from a register</td>
<td>$\text{Rd} \rightarrow \text{mem8}[\text{address}]$</td>
</tr>
<tr>
<td>LDRH</td>
<td>load halfword into a register</td>
<td>$\text{Rd} &lt;- \text{mem16}[\text{address}]$</td>
</tr>
<tr>
<td>STRH</td>
<td>save halfword into a register</td>
<td>$\text{Rd} \rightarrow \text{mem16}[\text{address}]$</td>
</tr>
<tr>
<td>LDRSB</td>
<td>load signed byte into a register</td>
<td>$\text{Rd} &lt;- \text{SignExtend (mem8}[\text{address}])$</td>
</tr>
<tr>
<td>LDRSH</td>
<td>load signed halfword into a register</td>
<td>$\text{Rd} &lt;- \text{SignExtend (mem16}[\text{address}])$</td>
</tr>
</tbody>
</table>
Single-Register Transfer

• Example: \texttt{LDR r0, [r1] / STR r0, [r1]}

• The LDR and STR instructions can load and store data on a boundary alignment \(\rightarrow\) the same as datatype size being loaded or stored

\[
\begin{align*}
\text{; load register r0 with the contents of} \\
\text{; the memory address pointed to by register} \\
\text{; r1.} \\
\text{; LDR} \\
\text{; r0, [r1] ; = LDR r0, [r1, #0]} \\
\text{; store the contents of register r0 to} \\
\text{; the memory address pointed to by} \\
\text{; register r1.} \\
\text{; STR} \\
\text{; r0, [r1] ; = STR r0, [r1, #0]} \\
\end{align*}
\]
Single-Register Transfer

• This example shows a load from a memory address contained in register $r1$, followed by a store back to the same address in memory
• The offset from register $r1$ is zero
• Register $r1$ is called the based address register
Single-Register Load-Store Addressing Modes

- The ARM instruction set provides different modes for addressing memory.
- These modes incorporate one of the indexing methods:
  1. Preindex with writeback
  2. Preindex
  3. Postindex

<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>

Note: ! indicates that the instruction writes the calculated address back to the base address register.
Single-Register Load-Store Addressing Modes

• Example: \texttt{LDR r0, [r1, #4]}! / \texttt{LDR r0, [r1, #4]} / \texttt{LDR r0, [r1], #4}

• Preindex with writeback calculates an address from a base register plus address offset and then updates that address base register with the new address

• Preindex offset is the same as preindex with writeback but does NOT update the address base register

• Postindex only updates the address base register after the address is used

• The preindex mode is useful for accessing an element in a data structure

• The postindex and preindex with writeback modes are useful for traversing an array
Single-Register Load-Store Addressing Modes

**PRE**

```
r0 = 0x00000000
r1 = 0x00090000
mem32[0x00009000] = 0x01010101
mem32[0x00009004] = 0x02020202
```

```
LDR r0, [r1, #4]!
```

Preindexing with writeback:

**POST(1)**

```
r0 = 0x02020202
r1 = 0x00009004
```

```
LDR r0, [r1], #4
```

**POST(2)**

```
r0 = 0x02020202
r1 = 0x00009000
```

```
LDR r0, [r1], #4
```

Postindexing:

**POST(3)**

```
r0 = 0x01010101
r1 = 0x00009004
```
Single-Register Load-Store Addressing Modes

<table>
<thead>
<tr>
<th>Addressing mode and index method</th>
<th>Addressing(^1) syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preindex with immediate offset</td>
<td>([Rn, # +/-\text{offset}_{12}])</td>
</tr>
<tr>
<td>Preindex with register offset</td>
<td>([Rn, +/-Rm])</td>
</tr>
<tr>
<td>Preindex with scaled register offset</td>
<td>([Rn, +/-Rm, \text{shift} #\text{shift}_{\text{imm}}])</td>
</tr>
<tr>
<td>Preindex writeback with immediate offset</td>
<td>([Rn, # +/-\text{offset}_{12}])!</td>
</tr>
<tr>
<td>Preindex writeback with register offset</td>
<td>([Rn, +/-Rm]!)</td>
</tr>
<tr>
<td>Preindex writeback with scaled register offset</td>
<td>([Rn, +/-Rm, \text{shift} #\text{shift}_{\text{imm}}]!)</td>
</tr>
<tr>
<td>Immediate postindexed</td>
<td>([Rn], # +/-\text{offset}_{12})</td>
</tr>
<tr>
<td>Register postindexed</td>
<td>([Rn], +/-Rm)</td>
</tr>
<tr>
<td>Scaled register postindexed</td>
<td>([Rn], +/-Rm, \text{shift} #\text{shift}_{\text{imm}})</td>
</tr>
</tbody>
</table>

- The addressing modes available with a particular load or store instruction depend on the instruction class.
- A signed offset or register is denoted by “+/-” \(\rightarrow\) can be positive or negative offset from the base address register \(Rn\).
Single-Register Load-Store Addressing Modes

- The base address register is a pointer to a byte in memory, and the offset specifies a number of bytes.
- **Immediate** means the address is calculated using the base address register and a 12-bit offset encoded in the instruction.
- **Register** means the address is calculated using the base address register and a specific register’s content.
- **Scaled** means the address is calculated using the base address register and a barrel shift operation.
Single-Register Load-Store Addressing Modes

- Example of \texttt{LDR} instructions using different addressing mode

<table>
<thead>
<tr>
<th>Instruction</th>
<th>$r0 =$</th>
<th>$r1 +=$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preindex with writeback</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR $r0,[r1,#0x4]$!</td>
<td>\texttt{mem32[r1+0x4]}</td>
<td>0x4</td>
</tr>
<tr>
<td>LDR $r0,[r1,r2]$!</td>
<td>\texttt{mem32[r1+r2]}</td>
<td>$r2$</td>
</tr>
<tr>
<td>LDR $r0,[r1,r2,LSR#0x4]$!</td>
<td>\texttt{mem32[r1+(r2 LSR 0x4)]}</td>
<td>(r2 LSR 0x4)</td>
</tr>
<tr>
<td>Preindex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR $r0,[r1,#0x4]$</td>
<td>\texttt{mem32[r1+0x4]}</td>
<td>not updated</td>
</tr>
<tr>
<td>LDR $r0,[r1,r2]$</td>
<td>\texttt{mem32[r1+r2]}</td>
<td>not updated</td>
</tr>
<tr>
<td>LDR $r0,[r1,-r2,LSR #0x4]$</td>
<td>\texttt{mem32[r1-(r2 LSR 0x4)]}</td>
<td>not updated</td>
</tr>
<tr>
<td>Postindex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR $r0,[r1],#0x4$</td>
<td>\texttt{mem32[r1]}</td>
<td>0x4</td>
</tr>
<tr>
<td>LDR $r0,[r1],r2$</td>
<td>\texttt{mem32[r1]}</td>
<td>$r2$</td>
</tr>
<tr>
<td>LDR $r0,[r1],r2,LSR #0x4$</td>
<td>\texttt{mem32[r1]}</td>
<td>(r2 LSR 0x4)</td>
</tr>
</tbody>
</table>
Single-Register Load-Store Addressing Modes

- Single register load-store addressing, halfword, signed halfword, signed byte and double word

<table>
<thead>
<tr>
<th>Addressing(^2) mode and index method</th>
<th>Addressing(^2) syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preindex immediate offset</td>
<td>[Rn, #+/-offset_8]</td>
</tr>
<tr>
<td>Preindex register offset</td>
<td>[Rn, +/-Rm]</td>
</tr>
<tr>
<td>Preindex writeback immediate offset</td>
<td>[Rn, #+/-offset_8]!</td>
</tr>
<tr>
<td>Preindex writeback register offset</td>
<td>[Rn, +/-Rm]!</td>
</tr>
<tr>
<td>Immediate postindexed</td>
<td>[Rn], #+/-offset_8</td>
</tr>
<tr>
<td>Register postindexed</td>
<td>[Rn], +/-Rm</td>
</tr>
</tbody>
</table>
Single-Register Load-Store Addressing Modes

- Variation of `STRH` instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result</th>
<th>$r1 + =$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preindex with writeback</td>
<td><code>STRH r0,[r1,#0x4]</code>!</td>
<td><code>mem16[r1+0x4]=r0</code></td>
</tr>
<tr>
<td>Preindex</td>
<td><code>STRH r0,[r1,r2]</code>!</td>
<td><code>mem16[r1+r2]=r0</code></td>
</tr>
<tr>
<td>Postindex</td>
<td><code>STRH r0,[r1],#0x4</code></td>
<td><code>mem16[r1]=r0</code></td>
</tr>
<tr>
<td></td>
<td><code>STRH r0,[r1],r2</code></td>
<td><code>mem16[r1]=r0</code></td>
</tr>
</tbody>
</table>
Multiple-Register Transfer

- Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction.
- The transfer occurs from a base address register Rn pointing into memory.
- Multiple-register transfer instructions are more efficient than single-register transfer for moving blocks of data around memory and saving and restoring context and stacks.
- Load-store multiple instruction can increase interrupt latency.
- ARM implementations do not usually interrupt instructions while they are executing.
- If an interrupt has been raised, then it has no effect until the load-store multiple instruction is complete.
Multiple-Register Transfer

- Compilers, such as `armcc`, provide a switch to control the maximum number of registers being transferred on a load-store → limits the maximum interrupt latency

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LDM</td>
<td>load multiple registers</td>
<td><code>{Rd}</code><em>N ← mem32[start address + 4</em>N] optional Rn updated</td>
</tr>
<tr>
<td>STM</td>
<td>save multiple registers</td>
<td><code>{Rd}</code><em>N → mem32[start address + 4</em>N] optional Rn updated</td>
</tr>
</tbody>
</table>

- There are also various addressing mode for the load-store multiple instructions
- Here N is the number of registers in the list registers
Multiple-Register Transfer

- Addressing mode for load-store multiple instructions

<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>

- Note that for all these four modes, the highest-numbered register always corresponds to the highest address in the memory
Multiple-Register Transfer

- Any subset of the current bank of registers can be transferred to memory or fetched from memory.
- The base register Rn determines the source or destination address for a load-store multiple instruction.
- This register can be optionally updated following the transfer → occurs when the register Rn is followed by the ! character (similar to the single-register load-store using preindex with writeback).
Multiple-Register Transfer

- Example: \texttt{LDM A r0!, \{r1-r3\}}
- In this example, register \texttt{r0} is the base register Rn and is followed by \texttt{!} indicating that the register is updated after the instruction is executed.
- Notice that within the load multiple instruction, the registers are not individual listed.
- The “-” character is used to identify a range of registers.
- In this example, the range is from register \texttt{r1} to \texttt{r3} inclusive.
- Each register can also be listed, using a comma to separate each register within “{“ and “}” brackets.
Multiple-Register Transfer

**PRE**

mem32[0x80018] = 0x03
mem32[0x80014] = 0x02
mem32[0x80010] = 0x01
r0 = 0x00080010
r1 = 0x00000000
r2 = 0x00000000
r3 = 0x00000000

LDMIA r0!, {r1-r3}

**POST**

r0 = 0x0008001c
r1 = 0x00000001
r2 = 0x00000002
r3 = 0x00000003
Multiple-Register Transfer

- The base register \( r0 \) points to memory address 0x80010 in the \texttt{PRE} condition
- Memory addresses 0x80010, 0x80014 and 0x80018 contain the values 1, 2, and 3 respectively

\[
\begin{array}{|c|c|}
\hline
\text{Address pointer} & \text{Memory address} & \text{Data} \\
\hline
0x80020 & 0x00000005 & \text{r3 = 0x00000000} \\
0x8001c & 0x00000004 & \text{r2 = 0x00000000} \\
0x80018 & 0x00000003 & \text{r1 = 0x00000000} \\
0x80014 & 0x00000002 & \hline
0x80010 & 0x00000001 & \hline
0x8000c & 0x00000000 & \\
\hline
\end{array}
\]

\texttt{PRE}-condition for \texttt{LDM} A instruction

\texttt{LDM} A instruction
Multiple-Register Transfer

- After the load multiple instruction executes register \( r1 \), \( r2 \), and \( r3 \) contain these values
- The base register \( r0 \) now points to memory address \( 0x8001c \) after the last loaded word

<table>
<thead>
<tr>
<th>Address Pointer</th>
<th>Memory Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r0 = 0x8001c )</td>
<td>0x80020</td>
<td>0x000000005</td>
</tr>
<tr>
<td></td>
<td>0x8001c</td>
<td>0x000000004</td>
</tr>
<tr>
<td></td>
<td>0x80018</td>
<td>0x000000003</td>
</tr>
<tr>
<td></td>
<td>0x80014</td>
<td>0x000000002</td>
</tr>
<tr>
<td></td>
<td>0x80010</td>
<td>0x000000001</td>
</tr>
<tr>
<td></td>
<td>0x8000c</td>
<td>0x000000000</td>
</tr>
</tbody>
</table>

**POST-condition for LDM A instruction**
Multiple-Register Transfer

- Now replace the `LDM A` instruction with a load multiple and increment before `LDM B` instruction and use the same `PRE` condition.
- The first word pointed to by register `r0` is ignored and register `r1` is

<table>
<thead>
<tr>
<th>Address pointer</th>
<th>Memory address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r0 = 0x8001c</code></td>
<td><code>0x80020</code></td>
<td><code>0x00000005</code></td>
</tr>
<tr>
<td></td>
<td><code>0x8001c</code></td>
<td><code>0x00000004</code></td>
</tr>
<tr>
<td></td>
<td><code>0x80018</code></td>
<td><code>0x00000003</code></td>
</tr>
<tr>
<td></td>
<td><code>0x80014</code></td>
<td><code>0x00000002</code></td>
</tr>
<tr>
<td></td>
<td><code>0x80010</code></td>
<td><code>0x00000001</code></td>
</tr>
<tr>
<td></td>
<td><code>0x8000c</code></td>
<td><code>0x00000000</code></td>
</tr>
</tbody>
</table>

**POST-condition for `LDM B` instruction**

- `r3 = 0x00000004`  
- `r2 = 0x00000003`  
- `r1 = 0x00000002`
Multiple-Register Transfer

- After execution, register $r0$ now points to the last loaded memory location.
- This is in contrast with \texttt{LDM} example, which pointed to the next memory location.
- The decrement versions \texttt{DA} and \texttt{DB} of the load-store multiple instructions decrement the start address and then store to ascending memory locations.
- This is equivalent to descending memory but accessing the register in reverse order.
- With the increment and decrement load multiples, you can access arrays of forwards or backwards.
- The also allow for stack push and pull operations.
Multiple-Register Transfer

- The following table shows a list of load-store multiple instruction pairs
- If a store with base update is used → the paired load instruction of the same number of registers will reload the data and restore the base address pointer
- This is useful when you need to temporarily save a group of registers and restore 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>
Multiple-Register Transfer

- Example: `STMIB r0!, {r1-r3} / LDMDA r0!, {r1-r3}`
- This example shows an **STM** increment before instruction followed by **LDM** decrement after instruction.
Multiple-Register Transfer

- The STMIB instruction stores the values 7, 8, 9 to memory → corrupt register \( r1 \) to \( r3 \)
- The LDMDA reloads the original values and restores the base pointer \( r0 \)
- The following code illustrates the use of the load-store multiple instructions with a block memory copy example
- This example is a simple routine that copies blocks of 32 bytes from a source address location to a destination address location
- The example has two load-store multiple instructions, which use the same increment after addressing mode
Multiple-Register Transfer

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

loop

; load 32 bytes from source and update r9 pointer
LDMIA  r9!, {r0-r7}

; store 32 bytes to destination and update r10 pointer
STMIA  r10!, {r0-r7} ; and store them

; have we reached the end
CMP     r9, r11
BNE     loop
Multiple-Register Transfer

- This routine relies on registers \( r9, r10, \) and \( r11 \) being set up before the code is executed.
- Register \( r9 \) and \( r11 \) determine the data to be copied, and register \( r10 \) points to the destination in memory for the data.
- \texttt{LDMIA} loads the data pointed by register \( r9 \) into registers \( r0 \) to \( r7 \).
- It also updates \( r9 \) to point to the next block of data to be copied.
- \texttt{STMIA} copies the contents of registers \( r0 \) to \( r7 \) to the destination memory address pointed to by register \( r10 \).
- It also updates \( r10 \) to point to the next destination location.
Multiple-Register Transfer

- **CMP** and **BNE** compare pointers $r9$ and $r11$ to check whether the end of the block copy has been reached.
- If the block copy is complete, then the routine finishes → otherwise the loop repeats with the updated values of register $r9$ and $r10$.
- The **BNE** is the branch instruction **B** with a condition mnemonic **NE** (not equal).
- If the previous compare instruction sets the condition flags to **not equal** → the branch instruction is executed.
- The following figure shows the memory map of the block memory copy and how the routine moves through memory.
Multiple-Register Transfer
Stack Operations

- The ARM architecture uses the load-store multiple instructions to carry out stack operations.
- The **pop** operation (removing data from a stack) uses a load multiple instruction.
- Similarly, the **push** operation (placing data onto the stack) uses a store multiple instruction.
- When using the stack, you have to decide whether the stack will grow up or down in memory.
- A stack is either ascending (A) or descending (D).
- Ascending stacks grow towards higher memory addresses.
- Descending stacks grow towards lower memory addresses.
Stack Operations

• When you use a full stack (F), the stack pointer sp points to an address that is the last used or full location
• If you use an empty stack (E) the sp points to an address that is the first unused or empty location
• There are a number of load-store multiple addressing mode aliases available to support 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>LMDMB</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>
Stack Operations

• Next to the pop column is the actual load multiple instruction equivalent
• For example, a full ascending stack would have the notation FA appended to the load multiple instruction → LDMFA
• This would be translated into an LDMDA instruction
• ARM has specified an ARM-Thumb Procedure Call Standard (ATPCS) that defines how routines are called and how registers are allocated
• In the ATPCS, stacks are defined as being full descending stacks → the LDMFD and STMFD instructions provide the pop and push functions, respectively
Stack Operations

• Example: `STMFD sp!, {r1, r4}`
• The `STMFD` instruction pushes registers onto the stack, updating `sp`
• The figure shows a push onto a full descending stack → when the stack grows the stack pointer points to the last full entry.

<table>
<thead>
<tr>
<th>PRE</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = 0x00000002</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4 = 0x00000003</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sp = 0x00080014</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>POST</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = 0x00000002</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4 = 0x00000003</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sp = 0x0008000c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stack Operations

• Example: \texttt{STMED sp!, \{r1, r4\}}

• The \texttt{STMED} instruction pushes registers onto the stack but update the register \texttt{sp} to point to the next empty location.
Stack Operations

- When handling a checked stack there are three attributes that need to be preserved
  1. Stack base
  2. Stack pointer
  3. Stack limit
- The **stack base** is the starting address of the stack in memory
- The **stack pointer** initially points to the stack base → as data is pushed onto the stack, the stack pointer descends memory and continuously points to the top of stack
- If the stack pointer passes the **stack limit** → then the **stack overflow error** has occurred
Stack Operations

- Stack base and stack pointer
Stack Operations

• The following is the code that checks for stack overflow errors for a descending stack;

```assembly
; check for stack overflow
SUB sp, sp, #size
CMP sp, r10
BLLO _stack_overflow ; condition
```

• ATPCS defines register \texttt{r10} as the stack limit or \texttt{sl}
• This is optional since it is only used when stack checking is enabled
Stack Operations

• The BLLO instruction is a branch with link instruction plus the condition mnemonic $LO$ (unsigned lower).

• If $sp$ is less than $r10$ after the new items are pushed onto the stack → stack overflow error has occurred.

• If $sp$ goes back past the stack base → then the stack underflow error has occurred.
Swap Instruction

- The swap instruction is a special case of a load-store instruction.
- It swaps the contents of memory with the contents of a register.
- This instruction is an atomic operation.
- Atomic operation means that it reads and writes a location in the same bus operation, preventing any other instruction from reading or writing to that location until it completes.
- Swap cannot be interrupted by any other instruction or any other bus access.
- Thus the system “holds the bus” until the transaction is complete.
### Swap Instruction

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Effects</th>
</tr>
</thead>
</table>
| SWP         | swap a word between memory and a register | \[
\begin{align*}
\text{tmp} &= \text{mem32}[Rn] \\
\text{mem32}[Rn] &= Rm \\
Rd &= \text{tmp}
\end{align*}
\] |
| SWPB        | swap a byte between memory and a register | \[
\begin{align*}
\text{tmp} &= \text{mem8}[Rn] \\
\text{mem8}[Rn] &= Rm \\
Rd &= \text{tmp}
\end{align*}
\] |

- **Example:** `SWP r0, r1, [r2]`
- The swap instruction loads a word from memory into register `r0` and overwrites the memory with register `r1`
Swap Instruction

From the syntax, this instruction can also have a byte size quantifier $B \rightarrow$ the instruction allows for both a word and a byte swap.
Swap Instruction

- Example: \( \text{SWP} \ r_3, \ r_2, \ [r_1] \)

- This example shows a simple data guard can be used to protect data from being written by another task.

- The \( \text{SWP} \) instruction “holds the bus” until the transaction is complete:

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

- The address pointed to by the semaphore either contains the value 0 or 1
- When semaphore equals 1 → the service in question is being used by another process
- The routine will continue to loop around until the service is released by the other process → when the semaphore address location contains the value 0
Software Interrupt Instruction

- A software interrupt instruction (SWI) causes a software interrupt exception → providing a mechanism for applications to call operating system routines.

> Syntax: SWI{<cond>} SWI_number

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

- When the processor executes an SWI instruction → it sets the program counter pc to the offset 0x8 in the vector table.
Software Interrupt Instruction

• The instruction also forces the processor mode to SVC → allows an operating system routine to be called in privileged mode
• Each \textit{SW} instruction has an associated \textit{SW} number → use to represent a particular function call or feature
• Example: 0x00008000 \textit{SW} 0x123456
• This example shows an \textit{SW} call with \textit{SW} number 0x123456
• Typically the \textit{SW} instruction is executed in user mode
Software Interrupt Instruction

- Since SWI instructions are used to call operating system routines, you need some form of parameter passing - using registers
- In this example, register r0 is used to pass the parameter 0x12
- The return values are also passed back via registers
Software Interrupt Instruction

- Code called the **SWI handler** is required to process the \texttt{sw} call
- The handler obtains the SWI number using the address of the executed instruction \rightarrow which calculated from the link register \texttt{lr}
- The SWI number is determined by
  \[
  \text{SWI\_Number} = \text{<SW\_instruction>} \text{ AND NOT } (0xff000000)
  \]
- Here the SWI instruction is the actual 32-bit SWI instruction executed by processor
Software Interrupt Instruction

- Example: This example shows the start of an SWI handler implementation.
- The code fragment determines what SWI number is being called and places that number into register $r10$.
- Note that the load instruction first copies the complete SWI instruction into register $r10$.
- The BIC instruction masks off the top bits of the instruction, leaving the SWI number.
- We assume the SWI has been called from ARM state.
- The number in register $r10$ is then used by the SWI handler to call the appropriate SWI service routine.
Software Interrupt Instruction

```
SWI_handler

; Store registers r0-r12 and the link register
STMFD sp!, {r0-r12, lr}

; Read the SWI instruction
LDR r10, [lr, #-4]

; Mask off top 8 bits
BIC r10, r10, #0xff000000

; r10 - contains the SWI number
BL serviceRoutine

; return from SWI handler
LDMFD sp!, {r0-r12, pc}^
```
Software Interrupt Instruction

On SWI, the processor:
1. copies CPSR to SPSR_SVC
2. sets the CPSR mode bits to supervisor mode
3. sets the CPSR IRQ to disable
4. stores the value (PC + 4) into LR_SVC
5. forces PC to 0x08

USER Program: 0x40002000
ADD r0, r0, r1
SWI 0x02
SUB r2, r2, r0

Warning:
What was in R0? User program may have been using this register. Therefore, cannot just use it - must first save it first (push stack)

ADD  r0,r0,r1
SWI  0x02
SUB  r2,r2,r0

Usage of BIC (Bit Clear)
E.g. BIC R0, R0, #%1011 ;
Clear bit-0, bit-1, bit-3 in R

Vector Table (spring board)
starting at 0x00 in memory

0x00 to R_Handler (Reset)
0x04 to U_Handler (Undef instr.)
0x08 to S_Handler (SWI)
0x0c to P_Handler (Prefetch abort)
0x10 to D_Handler (Data abort)
0x14 ... (Reserved)
0x18 to I_Handler (IRQ)
0x1c to F_Handler (FIQ)

LDR r0,[lr,--4]
BIC r0,r0,#0xff000000
// now the vector is in r0
switch (r0){
case 0x00: service_SWI1();
case 0x01: service_SWI2();
case 0x02: service_SWI3();
MOV r0, pc, lr
}

From
www.cs.ucr.edu/~amitra/context_switch/extra/04_swi.ppt
Program Status Register Instructions

- The ARM instruction set provides two instructions to directly control a program status register ($psr$)
- The $MRS$ instruction transfers the contents of either the $cpsr$ or $spsr$ into a register
- In the reverse direction, the $MSR$ instruction transfers the contents of a register into the $cpsr$ or $spsr$
- Together these instructions are used to read and write $cpsr$ and $spsr$
- In the syntax, a label called field can be any combination of control ($c$), extension ($x$), status ($s$), and flags ($f$)
- These fields relate to particular byte regions in a $psr$
Program Status Register Instructions

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

<table>
<thead>
<tr>
<th>MRS</th>
<th>copy program status register to a general-purpose register</th>
<th>Rd = psr</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR</td>
<td>move a general-purpose register to a program status register</td>
<td>psr[field] = Rm</td>
</tr>
<tr>
<td>MSR</td>
<td>move an immediate value to a program status register</td>
<td>psr[field] = immediate</td>
</tr>
</tbody>
</table>

Fields

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>31 30 29 28</td>
<td></td>
<td>7 6 5 4 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>V</th>
<th>I</th>
<th>F</th>
<th>T</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Program Status Register Instructions

• The `c` field controls the interrupt masks, Thumb state, and processor mode
• Example: Enable `IRQ` interrupt by clearing the `I` mask
• This operation involves using both the `MRS` and `MSR` instructions to read from and then write to the `cpsr`

```
PRE       cpsr = nzcvqIft_SVC
MRS       rl, cpsr
BIC       rl, rl, #0x80 ; 0b01000000
MSR       cpsr_c, rl
POST      cpsr = nzcvqIft_SVC
```
Program Status Register

Instructions

- The MRS first copies the cpsr into register \textit{r1}
- The BIC instruction clears bit 7 of \textit{r1}
- Register \textit{r1} is then copied back into the cpsr, which enables IRQ interrupts
- This code preserves all the other setting in the \textit{cpsr} and only modifies the \textit{I} bit in the control field (use \textit{cpsr\_c Control (Interrupt and modes)} instead of \textit{cpsr} )
- Other fields are \textit{cpsr\_s Status (unused)}, \textit{cpsr\_x Extended (unused)}, \textit{cpsr\_f Flags (condition codes)}
- This example is in SVC mode, in user mode you can read all \textit{cpsr} bits but you can only update the condition flag field \textit{f}
Coprocessor Instructions

- Coprocessor instructions are used to extend the instruction set
- A coprocessor can either provide additional computation capability or be used to control the memory subsystem including caches and memory management
- The coprocessor instructions include data processing, register transfer, and memory transfer instructions
- This is only short overview and these instructions are only used by cores with a coprocessor

Syntax: 
- CDP{<cond>} cp, opcode1, Cd, Cn {, opcode2}
- <MRC|MCR>{<cond>} cp, opcode1, Rd, Cn, Cm {, opcode2}
- <LDC|STC>{<cond>} cp, Cd, addressing
**Coproprocessor Instructions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDP</td>
<td>coprocessor data processing—perform an operation in a coprocessor</td>
</tr>
<tr>
<td>MRC</td>
<td>coprocessor register transfer—move data to/from coprocessor registers</td>
</tr>
<tr>
<td>MCR</td>
<td>coprocessor memory transfer—load and store blocks of memory to/from a coprocessor</td>
</tr>
</tbody>
</table>

- The *cp* field represents the coprocessor number between *p0* and *p15*
- The *opcode* fields describe the operation to take place on the coprocessor
- The *Cn*, *Cm*, and *Cd* fields describe registers within coprocessor
- The coprocessor operations and registers depend on the specific coprocessor you are using
- Coprocessor 15 (C15) is reserved for system control purposes, e.g. memory management, write buffer control, cache control, and identification registers
Coproces sor Instructions

• This example shows a CP15 register being copied into a general-purpose register

  ; transferring the contents of CP15 register c0 to register r10
  MRC p15, 0, r10, c0, c0, 0

• Here CP15 register 0 contains the processor identification number

• This register is copied into the general-purpose register r10

• CP15 configures the processor core and has a set of dedicated register to store configuration information

• A value written into a register sets a configuration attribute → e.g. switch on the cache
Coprocessor Instructions

- CP15 is called the system control processor
- Both \texttt{MRC} and \texttt{MCR} instructions are used to read and write to CP15
- Register \textit{Rd} is the core destination register, \textit{Cn} is the primary register, \textit{Cm} is the secondary register and \textit{opcode2} is a secondary register (extended register) modifier
- In the example, the instruction is used to move the contents of CP15 control register \textit{c1} into register \textit{r1} of the processor core:
  \[ \texttt{MRC p15, 0, r1, c1, c0, 0} \]
- The shorthand notation for CP15 uses the following format
  \[ \texttt{CP15:cX:cY:Z} \]
Coprocessor Instructions

• CP15: defines it as a coprocessor 15
• cX: the primary register, the primary register X can have a value between 0 and 15
• cY: the secondary or extend register, the secondary register Y can have a value between 0 and 15
• opcode2: is an instruction modifier and can have a value between 0 and 7
• Some operations may also use a nonzero value w of opcode1 → can be written as

  CP15:w:cX:cY:Z
Loading Constants

- Note that there is no ARM instruction to move a 32-bit constant into a register.
- ARM instructions are 32 bits in size → cannot specify a general 32-bit constant.
- Because some of those bits are needed to identify the instruction’s opcode, others are used to indicate the destination register → leaving fewer than 32 bits left for literal number.
- ARM uses two pseudo-instruction to move a 32-bit value into a register.

**Syntax:**

```
LDR Rd, =constant
ADR Rd, label
```

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

- The pseudo-instruction does not actually exist → the assembler translates it into one or more real instructions
- The pseudo-instruction writes a 32-bit constant to a register using whatever instructions are available
- It defaults to a memory read if the constant cannot be encoded using other instructions
- The second pseudo-instruction writes a relative address into a register → which will be encoded using a pc-relative expression
Loading Constants

- Example: This example shows an LDR instruction loading a 32-bit constant 0xff00ffff into register \( r0 \)
- It involves a memory access to load the constant \( \rightarrow \) this can be expensive for time-critical routines

\[
\begin{align*}
\text{LDR} & \quad r0, [pc, \#\text{constant_number}-8-\{PC}\} \\
\text{DCD} & \quad 0xff00ffff
\end{align*}
\]
Loading Constants

• Example: This example shows an alternative method to load the same constant into register \texttt{r0} by using an \texttt{MVN} instruction

\begin{verbatim}
PRE    none...

MVN    r0, \#0x00ff0000

POST   r0 = 0xff00ffff
\end{verbatim}

• Thus there are alternatives to accessing memory, but they depend upon the constant you are trying to load
• Compilers and assemblers use clever techniques to avoid loading a constant from memory
Loading Constants

• These tools have algorithms to find the optimal number of instructions required to generate a constant in a register and make extensive use of the barrel shifter.
• If the tools cannot generate the constant by these methods → then it is loaded from memory.
• The `LDR` pseudo-instruction either inserts and `MOV` or `MVN` instruction to generate a value (if possible).
• Or generates an `LDR` instruction with `pc-relative address to read the constant from a literal pool` (a data area embedded within the code).
Loading Constants

The above table shows pseudo-code conversions:

- The first conversion produces a simple `MOV` instruction.
- The second conversion produces a `pc`-relative load.
- It is recommended that you use this pseudo-instruction to load a constant.

Another useful pseudo-instruction is the `ADR` instruction, or address relative → it places the address of the given label into register Rd, using a `pc`-relative add or subtract.

<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>
ARMv5E Extensions

- The ARMv5E extensions provide many new instructions.
- One of the most important additions is the signed multiply accumulate instructions that operate on 16-bit data.
- These operations are single cycle on many ARMv5E implementation.
- ARMv5E provides greater flexibility and efficiency when manipulating 16-bit values → important for applications such as 16-bit audio processing.
- The list of new instruction is shown in the following table.
## ARMv5E Extensions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLZ {&lt;cond&gt;} Rd, Rm</td>
<td>count leading zeros</td>
</tr>
<tr>
<td>QADD {&lt;cond&gt;} Rd, Rm, Rn</td>
<td>signed saturated 32-bit add</td>
</tr>
<tr>
<td>QDADD {&lt;cond&gt;} Rd, Rm, Rn</td>
<td>signed saturated double 32-bit add</td>
</tr>
<tr>
<td>QDSUB {&lt;cond&gt;} Rd, Rm, Rn</td>
<td>signed saturated double 32-bit subtract</td>
</tr>
<tr>
<td>QSUB {&lt;cond&gt;} Rd, Rm, Rn</td>
<td>signed saturated 32-bit subtract</td>
</tr>
<tr>
<td>SMLAxy {&lt;cond&gt;} Rd, Rm, Rs, Rn</td>
<td>signed multiply accumulate 32-bit (1)</td>
</tr>
<tr>
<td>SMLALxy {&lt;cond&gt;} RdLo, RdHi, Rm, Rs</td>
<td>signed multiply accumulate 64-bit</td>
</tr>
<tr>
<td>SMLAWy {&lt;cond&gt;} Rd, Rm, Rs, Rn</td>
<td>signed multiply accumulate 32-bit (2)</td>
</tr>
<tr>
<td>SMULxy {&lt;cond&gt;} Rd, Rm, Rs</td>
<td>signed multiply (1)</td>
</tr>
<tr>
<td>SMULWy {&lt;cond&gt;} Rd, Rm, Rs</td>
<td>signed multiply (2)</td>
</tr>
</tbody>
</table>
ARMv5E Extensions (Clear Leading Zero)

• The count leading zeros instruction counts the number of zeros between the most significant bit and the first bit set to 1

• Example: In this example, the first bit set to 1 has 27 zeros preceding it

• CLZ is useful in routines that have to normalize number

```
PRE
  r1 = 0b0000000000000000000000000000010000

CLZ r0, r1

POST
  r0 = 27
```
ARMv5E Extensions (Saturated Arithmetic)

• Normal ARM arithmetic instructions wrap around when you overflow an integer value → e.g. 0xffffffff + 1 = -0x80000000
• Thus when design an algorithm, you have to be careful not to exceed the maximum representable value in a 32-bit integer
• Example: This example shows what happens when the maximum value is exceeded

```
PRE       cpsr = nzcvqifiT_SVC
          r0 = 0x00000000
          r1 = 0x70000000 (positive)
          r2 = 0xffffffff (positive)
ADD      r0, r1, r2

POST      cpsr = NzcVqifiT_SVC
          r0 = 0xffffffff (negative)
```
ARMv5E Extensions (Saturated Arithmetic)

- In this example, register $r1$ and $r2$ contain positive numbers.
- Register $r2$ is equal to $0x7fffffff$ → the maximum positive value you can store in 32-bit.
- In perfect world adding these numbers together would result in a large positive number.
- Instead, the value becomes negative and the overflow flag, $V$, is set.
- But when using ARMv5E instructions you can saturate the result → once the highest number is exceeded the results remain at the maximum value of $0x7fffffff$.
- This avoids the requirement for any additional code to check for possible overflow.
ARMv5E Extensions (Saturated Arithmetic)

- The list of saturate instruction is as follows

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Saturated calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>QADD</td>
<td>Rd = Rn + Rm</td>
</tr>
<tr>
<td>QDADD</td>
<td>Rd = Rn + (Rm*2)</td>
</tr>
<tr>
<td>QSUB</td>
<td>Rd = Rn - Rm</td>
</tr>
<tr>
<td>QDSUB</td>
<td>Rd = Rn - (Rm*2)</td>
</tr>
</tbody>
</table>

- Example: This example shows the same data being passed into the QADD instruction

```
PRE
cpsr = nzcvqiFt_SVC
r0 = 0x00000000
r1 = 0x70000000 (positive)
r2 = 0x7ffffffff (positive)

QADD    r0, r1, r2

POST
cpsr = nzcvQiFt_SVC
r0 = 0x7ffffffff
```
ARMv5E Extensions (Saturated Arithmetic)

- Notice that the saturated number is return in register \textit{r0}
- Also the \textit{Q} bit (bit 27 of the \textit{cpsr}) has been set \rightarrow indicating saturation has occurred
- The \textit{Q} flag is sticky and will remain set until explicitly clear
ARMv5E Extensions (Multiply Instructions)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Signed Multiply [Accumulate]</th>
<th>Signed result</th>
<th>Q flag updated</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMLAxy</td>
<td>(16-bit *16-bit)+ 32-bit</td>
<td>32-bit</td>
<td>yes</td>
<td>$Rd = (Rm.x * Rs.y) + Rn$</td>
</tr>
<tr>
<td>SMLALxy</td>
<td>(16-bit *16-bit)+ 64-bit</td>
<td>64-bit</td>
<td>—</td>
<td>$[RdHi, RdLo] += Rm.x * Rs.y$</td>
</tr>
<tr>
<td>SMLAWy</td>
<td>(32-bit *16-bit) $\gg$ 16)+ 32-bit</td>
<td>32-bit</td>
<td>yes</td>
<td>$Rd = ((Rm * Rs.y) \gg 16) + Rn$</td>
</tr>
<tr>
<td>SMULxy</td>
<td>(16-bit *16-bit)</td>
<td>32-bit</td>
<td>—</td>
<td>$Rd = Rm.x * Rs.y$</td>
</tr>
<tr>
<td>SMULWy</td>
<td>(32-bit *16-bit) $\gg$ 16)</td>
<td>32-bit</td>
<td>—</td>
<td>$Rd = (Rm * Rs.y) \gg 16$</td>
</tr>
</tbody>
</table>

- The above table shows a complete list of the ARMv5E multiply instruction
- $x$ and $y$ select which 16 bits of a 32-bit register are used for the first and second operands, respectively
ARMv5E Extensions (Multiply Instructions)

- These x and y fields are set to a letter T for the top 16-bits, or the letter B for the bottom 16 bits
- For multiply accumulate operations with 32-bit results → Q flag indicates if the accumulate overflowed a signed 32-bit value
- Example: this example uses a signed multiply accumulate instruction SMLATB

PRE
\[
\begin{align*}
  r1 & = 0x20000001 \\
  r2 & = 0x20000001 \\
  r3 & = 0x00000004 \\
\end{align*}
\]

SMLATB r4, r1, r2, r3

POST
\[
  r4 = 0x00002004
\]
ARMv5E Extensions (Multiply Instructions)

• The instruction multiplies the top 16 bits of register \( r1 \) by the bottom 16 bits of register \( r2 \)
• It adds the result to register \( r3 \) and writes it to destination register \( r4 \)
Conditional Execution

• Most ARM instructions are **conditionally executed** → you can specify that the instruction only executes if the condition code flags pass a given condition or test.
• By using conditional execution instructions → can increase performance and code density.
• The condition field is a two-letter mnemonic appended to the instruction mnemonic.
• The default mnemonic is `AL`, or **always execute**.
• Conditional execution reduces the number of branches → also reduces the number of pipeline flushes → thus improves the performance of the executed code.
Conditional Execution

• Conditional execution depends upon two components
  1. Condition field
  2. Condition flags
• The condition field is located in the instruction, and the condition flags are located in the `cpsr`
• Example: The example shows an `ADD` instruction with the `EQ` condition appended
• This instruction will only be executed when the zero flag in the `cpsr` is set to 1
• Only comparison instruction and data processing instructions with the `s` suffix appended to the mnemonic update the condition flag in `cpsr`

; r0 = r1 + r2 if zero flag is set
ADDEQ r0, r1, r2
Conditional Execution

- Example: The following simple C code fragment will be compared with the assembler output using nonconditional and conditional instructions

```c
while (a!=b)
{
    if (a>b) a -= b; else b -= a;
}
```

- Let register \( r1 \) represent \( a \) and register \( r2 \) represent \( b \)
- The following code fragment shows the same algorithm written in ARM assembler
- This example only uses conditional execution on the branch instructions
Conditional Execution

; Greatest Common Divisor Algorithm

gcd

CMP r1, r2
BEQ complete
BLT lessthan
SUB r1, r1, r2
B gcd

less than

SUB r2, r2, r1
B gcd

complete

...
Conditional Execution

• Now compare the same code with full conditional execution → this dramatically reduces the number of instructions:

```
gcd
CMP      r1, r2
SUBGT    r1, r1, r2
SUBLT    r2, r2, r1
BNE      gcd
```