Hardware/Software Interface

Uwe R. Zimmer - The Australian National University
References for this chapter

[Patterson17]
David A. Patterson & John L. Hennessy
Computer Organization and Design – The Hardware/Software Interface
Chapter 2 “Instructions: Language of the Computer” & Chapter 3 “Arithmetic for Computers”
ARM edition, Morgan Kaufmann 2017
**Adding the value of two registers**

The CPU will fetch the content of the memory cell which PC is pointing to.

☞ We want the CPU to execute:

\[ r4 := r2 + r3 \]

☞ What to store in this memory cell?
Adding the value of two registers

Register bank

<table>
<thead>
<tr>
<th>r0</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>r4</th>
<th>r5</th>
<th>r6</th>
<th>r7</th>
</tr>
</thead>
</table>

| r8 | r9 | r10 | r11 | r12 | SP | LR | PC |

Status flags

ALU

NZCVQ

ADD $<Rd>, <Rn>, <Rm>$

```
0 0 0 1 1 0 0
```

Op Code Arguments
**Hardware/Software Interface**

**Adding the value of two registers**

Register bank

- r0
- r1
- r2
- r3
- r4
- r5
- r6
- r7
- r8
- r9
- r10
- r11
- r12
- SP
- LR
- PC

Status flags set:
- N Negative (MSB = 1)
- Z Zero (all bits zero)
- C Carry (carry out)
- V Overflow (sign wrong)

**Assembler**

```
ADD r4, r2, r3
```

```
0 0 0 1 1 0 0 0 1 1 0 1 0 1 0 0
```

- 16#18#
- 16#D4#

**Disassembler**

```
r4 := r2 + r3
```
Adding the value of two registers

Register bank

<table>
<thead>
<tr>
<th>r0</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>r4</th>
<th>r5</th>
<th>r6</th>
<th>r7</th>
</tr>
</thead>
<tbody>
<tr>
<td>r8</td>
<td>r9</td>
<td>r10</td>
<td>r11</td>
<td>r12</td>
<td>SP</td>
<td>LR</td>
<td>PC</td>
</tr>
</tbody>
</table>

Status flags

ALU

NZCVQ

ANDS <Rdn>, <Rm>

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rdn</td>
</tr>
</tbody>
</table>

Op Code  Arguments
Adding the value of two registers

**Register bank**

<table>
<thead>
<tr>
<th>r0</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>r4</th>
<th>r5</th>
<th>r6</th>
<th>r7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r8</th>
<th>r9</th>
<th>r10</th>
<th>r11</th>
<th>r12</th>
<th>sp</th>
<th>lr</th>
<th>pc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Status flags**

**ALU**

**NZCVQ**

---

**Assembler**

**Disassembler**

ANDS r5, r6

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

16#40# 16#35#

```
r5 := r5 & r6
```
Adding the value of two registers

Register bank

Status flags set:
- N Negative (MSB = 1)
- Z Zero (all bits zero)
- C Carry (carry out)
- V Overflow (sign wrong)

ADDS r4, r2, r3

```
0 0 0 1 1 0 0 0 1 1 0 1 0 1 0 0
```

16#18#
16#D4#

r4 := r2 + r3
ARM v7-M 32 bit add instructions

**add**{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}

**adc**{s}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}

**add**{s}<c><q> {<Rd>,} <Rn>, #<const>

**adc**{s}<c><q> {<Rd>,} <Rn>, #<const>

**qadd**<c><q> {<Rd>,} <Rn>, <Rm>

- **s**: sets the flags based on the result
- **c**: makes the command conditional. <c> can be EQ (equal), NE (not equal), CS (carry set), CC (carry clear), MI (minus), PL (plus), VS (overflow set), VC (overflow clear), HI (unsigned higher), LS (unsigned lower or same), GE (signed greater or equal), LT (signed less), GT (signed greater), LE (signed less or equal), AL (always)
- **q**: instruction width. Can be .N for narrow (16 bit) or .W for wide (32 bit)
- **Rd, Rn, Rm**: any register, incl. SP, LR and PC (with some restrictions). Result goes to Rn (if no Rd).
- **shift**: value of Rm is preprocessed with LSL (logical shift left – fills zeros), LSR (logical shift right – fills zeros), ASR (arithmetic shift right – keeps sign) or ROR (rotate right) followed by the #number of bits to shift/rotate by. There is also a RRX (rotate right by one incl. carry flag)
- **const**: an immediate value in the range 0..4095 directly or in the range 0..255 with rotation.
ARM v7-M 32 bit add instructions

\[\text{add}\{s\}<c><q> \{<Rd>,\} <Rn>, <Rm> \{,<shift>\}\]
\[\text{adc}\{s\}<c><q> \{<Rd>,\} <Rn>, <Rm> \{,<shift>\}\]
\[\text{add}\{s\}<c><q> \{<Rd>,\} <Rn>, \#<\text{const}>\]
\[\text{adc}\{s\}<c><q> \{<Rd>,\} <Rn>, \#<\text{const}>\]
\[\text{qadd}<c><q> \{<Rd>,\} <Rn>, <Rm>\]

- **s**: sets the flags based on the result
- **c**: makes the command conditional. `<c>` can be EQ (equal), NE (not equal), CS (carry set), CC (carry clear), MI (minus), PL (plus), VS (overflow set), VC (overflow clear), HI (unsigned higher), LS (unsigned lower or same), GE (signed greater or equal), LT (signed less), GT (signed greater), LE (signed less or equal), AL (always)
- **q**: instruction width. Can be `.N` for narrow (16 bit) or `.W` for wide (32 bit)
- **Rd, Rn, Rm**: any register, incl. SP, LR and PC (with some restrictions). Result goes to Rn (if no Rd).
- **shift**: value of Rm is preprocessed with LSL (logical shift left – fills zeros), LSR (logical shift right – fills zeros), ASR (arithmetic shift right – keeps sign) or ROR (rotate right) followed by the #number of bits to shift/rotate by. There is also a RRX (rotate right by one incl. carry flag)
- **const**: an immediate value in the range 0..4095 directly or in the range 0..255 with rotation.

Any of those instructions requires exactly one CPU cycle (in terms of throughput).

“Reduced Instruction Set Computing (RISC)”
Numeric CPU status flags

Natural binary numbers

$$0 \quad a+b \quad a \quad b \quad 2^{n-1}$$

Carry

Wrap-around or modulo $$2^n$$

$$a+b$$

$$2^n-1$$

Which of those operations will set which flag?

2's complement binary numbers

$$-2^{n-1} \quad c \quad a+b \quad 2d \quad d \quad 0 \quad 2c \quad a \quad b \quad 2^{n-1}-1$$

Overflow

Wrap-around

Overfl ow

Saturate

$$2c \quad c \quad 0 \quad a \quad b \quad a+b$$

Saturate

adds
adcs

qadd
ARM v7-M 32 bit Addition, Subtraction instructions

\[
\begin{align*}
\text{add}\{s\}<c><q>\ \{<Rd>,\} \ <Rn>, \ <Rm> \ \{,<shift>\} & \quad ; \ Rd := Rn + Rm(\text{shifted}) \\
\text{adc}\{s\}<c><q>\ \{<Rd>,\} \ <Rn>, \ <Rm> \ \{,<shift>\} & \quad ; \ Rd := Rn + Rm(\text{shifted}) + C \\
\text{add}\{s\}<c><q>\ \{<Rd>,\} \ <Rn>, \ #<\text{const}> & \quad ; \ Rd := Rn + #<\text{const}> \\
\text{adc}\{s\}<c><q>\ \{<Rd>,\} \ <Rn>, \ #<\text{const}> & \quad ; \ Rd := Rn + #<\text{const}> + C \\
\text{qadd}<c><q>\ \{<Rd>,\} \ <Rn>, \ <Rm> & \quad ; \ Rd := Rn + Rm \ ; \text{ saturated} \\
\text{sub}\{s\}<c><q>\ \{<Rd>,\} \ <Rn>, \ <Rm> \ \{,<shift>\} & \quad ; \ Rd := Rn - Rm(\text{shifted}) \\
\text{sbc}\{s\}<c><q>\ \{<Rd>,\} \ <Rn>, \ <Rm> \ \{,<shift>\} & \quad ; \ Rd := Rn - Rm(\text{shifted}) - \text{NOT (C)} \\
\text{rsb}\{s\}<c><q>\ \{<Rd>,\} \ <Rn>, \ <Rm> \ \{,<shift>\} & \quad ; \ Rd := Rm(\text{shifted}) - Rn \\
\text{sub}\{s\}<c><q>\ \{<Rd>,\} \ <Rn>, \ #<\text{const}> & \quad ; \ Rd := Rn - #<\text{const}> \\
\text{sbc}\{s\}<c><q>\ \{<Rd>,\} \ <Rn>, \ #<\text{const}> & \quad ; \ Rd := Rn - #<\text{const}> - \text{NOT (C)} \\
\text{rsb}\{s\}<c><q>\ \{<Rd>,\} \ <Rn>, \ #<\text{const}> & \quad ; \ Rd := #<\text{const}> - Rn \\
\text{qsub}<c><q>\ \{<Rd>,\} \ Rn, \ Rm & \quad ; \ Rd := Rn - Rm \ ; \text{ saturated}
\end{align*}
\]

All instructions operate on 32bit wide numbers.

... versions for narrower numbers, as well as versions which operate on multiple narrower numbers in parallel exist as well.
64 bit Addition, Subtraction

As your registers are 32 bit wide, you need two steps to add two 64 bit numbers in \( r_3 : r_2, r_5 : r_4 \) (with \( r_2 \) and \( r_4 \) being the lower 32 bits) to one 64 bit number in \( r_1 : r_0 \):

\[
\begin{align*}
\text{adds} & \quad r_0, r_2, r_4 & ; \; r_0 & := r_2 + r_4 & \text{add least significant words, set flags} \\
\text{adcs} & \quad r_1, r_3, r_5 & ; \; r_1 & := r_3 + r_5 + C & \text{add most significant words and carry bit}
\end{align*}
\]

… and symmetrically if you need a 64 bit subtraction:

\[
\begin{align*}
\text{subs} & \quad r_0, r_2, r_4 & ; \; r_0 & := r_2 - r_4 & \text{least significant words, set flags} \\
\text{sbscs} & \quad r_1, r_3, r_5 & ; \; r_1 & := r_3 - r_5 - \text{NOT} (C) & \text{most significant words and carry bit}
\end{align*}
\]
ARM v7-M 32 bit Boolean (bit-wise) instructions

\[
\begin{align*}
\text{and}\{s\}<c><q> \{<Rd>,} <Rn>, <Rm> \{,<shift>\}; & \quad Rd := Rn \land Rm^{\text{shifted}} \\
\text{bic}\{s\}<c><q> \{<Rd>,} <Rn>, <Rm> \{,<shift>\}; & \quad Rd := Rn \land Rm^{\text{shifted}} \\
\text{orr}\{s\}<c><q> \{<Rd>,} <Rn>, <Rm> \{,<shift>\}; & \quad Rd := Rn \lor Rm^{\text{shifted}} \\
\text{orn}\{s\}<c><q> \{<Rd>,} <Rn>, <Rm> \{,<shift>\}; & \quad Rd := Rn \lor Rm^{\text{shifted}} \\
\text{eor}\{s\}<c><q> \{<Rd>,} <Rn>, <Rm> \{,<shift>\}; & \quad Rd := Rn \land \neg Rm^{\text{shifted}} \\
\text{and}\{s\}<c><q> \{<Rd>,} <Rn>, \#<\text{const}> & \quad Rd := Rn \land \text{const} \\
\text{bic}\{s\}<c><q> \{<Rd>,} <Rn>, \#<\text{const}> & \quad Rd := Rn \land \text{const} \\
\text{orr}\{s\}<c><q> \{<Rd>,} <Rn>, \#<\text{const}> & \quad Rd := Rn \lor \text{const} \\
\text{orn}\{s\}<c><q> \{<Rd>,} <Rn>, \#<\text{const}> & \quad Rd := Rn \lor \text{const} \\
\text{eor}\{s\}<c><q> \{<Rd>,} <Rn>, \#<\text{const}> & \quad Rd := Rn \lor \text{const} \\
\text{cmp}\{c\}<c><q> \{<Rn>,} <Rm> \{,<shift>\} ; & \quad \left(Rn - Rm^{\text{shifted}}\right) \rightarrow \text{Flags} \\
\text{cmn}\{c\}<c><q> \{<Rn>,} <Rm> \{,<shift>\} ; & \quad \left(Rn + Rm^{\text{shifted}}\right) \rightarrow \text{Flags} \\
\text{tst}\{c\}<c><q> \{<Rn>,} <Rm> \{,<shift>\} ; & \quad \left(Rn \land Rm^{\text{shifted}}\right) \rightarrow \text{Flags} \\
\text{teq}\{c\}<c><q> \{<Rn>,} <Rm> \{,<shift>\} ; & \quad \left(Rn \lor Rm^{\text{shifted}}\right) \rightarrow \text{Flags} \\
\text{cmp}\{c\}<c><q> \{<Rn>,} \#<\text{const}} ; & \quad \left(Rn - \text{const}\right) \rightarrow \text{Flags} \\
\text{cmn}\{c\}<c><q> \{<Rn>,} \#<\text{const}} ; & \quad \left(Rn + \text{const}\right) \rightarrow \text{Flags} \\
\text{tst}\{c\}<c><q> \{<Rn>,} \#<\text{const}} ; & \quad \left(Rn \land \text{const}\right) \rightarrow \text{Flags} \\
\text{teq}\{c\}<c><q> \{<Rn>,} \#<\text{const}} ; & \quad \left(Rn \lor \text{const}\right) \rightarrow \text{Flags}
\end{align*}
\]

This exhausts the simple ALU from chapter 1 …
**Hardware/Software Interface**

**ARM v7-M Move data inside the CPU**

```
mov{s}<c><q> <Rd>, <Rm>  ; Rd := Rm
mov{s}<c><q> <Rd>, #<const>  ; Rd := const

lsr{s}<c><q> <Rd>, <Rm>, #<n>
lsr{s}<c><q> <Rd>, <Rm>, <Rs>

asr{s}<c><q> <Rd>, <Rm>, #<n>
asr{s}<c><q> <Rd>, <Rm>, <Rs>

lsl{s}<c><q> <Rd>, <Rm>, #<n>
lsl{s}<c><q> <Rd>, <Rm>, <Rs>

ror{s}<c><q> <Rd>, <Rm>, #<n>
ror{s}<c><q> <Rd>, <Rm>, <Rs>

rrx{s}<c><q> <Rd>, <Rm>
```
ARM v7-M Move data inside the CPU

mov{s}<c><q> <Rd>, <Rm> ; Rd := Rm
mov{s}<c><q> <Rd>, #<const> ; Rd := const

lsr{s}<c><q> <Rd>, <Rm>, #<n> ;
lsr{s}<c><q> <Rd>, <Rm>, <Rs>

asr{s}<c><q> <Rd>, <Rm>, #<n> ;
asr{s}<c><q> <Rd>, <Rm>, <Rs>

ls1{s}<c><q> <Rd>, <Rm>, #<n> ;
ls1{s}<c><q> <Rd>, <Rm>, <Rs>

ror{s}<c><q> <Rd>, <Rm>, #<n> ;
ror{s}<c><q> <Rd>, <Rm>, <Rs>

rrx{s}<c><q> <Rd>, <Rm>

If this is numbers then ...

Rs/2^n rounded towards −∞

for 2’s complements

Rm · 2^n

© 2021 Uwe R. Zimmer, The Australian National University
Simple arithmetic inside the CPU

Calculate:

\[ e := a + b - 2c \]

assuming all types are 32bit 2’s complement numbers (Integer),
r1 holds a, r2 holds b, r3 holds c, and the results should be in r4.
Simple arithmetic inside the CPU

Calculate:

\[ e := a + b - 2c \]

assuming all types are 32-bit 2’s complement numbers (Integer),
r1 holds a, r2 holds b, r3 holds c, and the results should be in r4.

```
add r5, r1, r2
lsl r6, r3, #1  ; you could also write: mov r6, r3, lsl #1
sub r4, r5, r6
```

We need temporary storage (r5, r6) in the process as we didn’t want to overwrite the original values. Yet the total number of registers is always limited.
Calculate:

\[ e := a + b - 2c \]

assuming all types are 32 bit 2’s complement numbers (Integer), r1 holds a, r2 holds b, r3 holds c, and the results should be in r4.

\[
\begin{align*}
    & \text{add} & r5, r1, r2 \\
    & \text{lsl} & r6, r3, #1 ; \text{you could also write: mov } r6, r3, \text{lsl } #1 \\
    & \text{sub} & r4, r5, r6 \\
\end{align*}
\]

We need temporary storage (r5, r6) in the process as we didn’t want to overwrite the original values. Yet the total number of registers is always limited.

How about we assume that values are no longer needed after this expression:
Calculate:
\[ e := a + b - 2c \]
assuming all types are 32 bit 2’s complement numbers (Integer), 
r1 holds a, r2 holds b, r3 holds c, and the results should be in r4.

\[
\begin{align*}
\text{add} & \quad r5, r1, r2 \\
\text{lsl} & \quad r6, r3, #1 \quad ; \quad \text{you could also write: mov } r6, r3, lsl \ #1 \\
\text{sub} & \quad r4, r5, r6
\end{align*}
\]

We need temporary storage \((r5, r6)\) in the process as we didn’t want to over-write the original values. Yet the total number of registers is always limited.

How about we assume that values are no longer needed after this expression:

\[
\begin{align*}
\text{add} & \quad r1, r1, r2 \\
\text{lsl} & \quad r3, r3, #1 \\
\text{sub} & \quad r4, r1, r3
\end{align*}
\]

… your compiler will know when such side-effects are ok and when not.

Any overflows?
Simple arithmetic inside the CPU

Calculate:

\[ e := a + b - 2c \]

We need to check results after each step:

- `adds r1, r1, r2` ; need to check overflow flag
- `lsl r3, r3, #1` ; need to check that the sign did not change
- `subs r4, r1, r3` ; need to check overflow flag again

↑ We don’t have the means yet to branch off into different actions in case things go bad ... to come soon.
Calculate:

\[ e := a + b - 2c \]

We need to check results after each step:

- `adds r1, r1, r2 ; need to check overflow flag`
- `lsl r3, r3, #1 ; need to check that the sign did not change`
- `subs r4, r1, r3 ; need to check overflow flag again`

⚠️ We don’t have the means yet to branch off into different actions in case things go bad … to come soon.

Or we use saturation arithmetic and live with the error:

- `qadd r1, r1, r2`
- `qadd r3, r3, r3`
- `qsub r4, r1, r3`

⚠️ If we know we need to carry on either way, this at least minimizes the local errors.
Cortex-M4 Address Space

Your CPU has 32 bit of address space

\[ 4 \text{ GB} \]

... address space does not equate to physical memory!

Not all memory is equal: Some memory ...

... can be executed
... can be written to or read from or both
... has side-effects (coffee cups fall over)
... has strictly-ordered access
... does not physically exist
In its most basic form the value of a register is interpreted as an address and the memory content there is loaded into another register.
Yet: most data is structured.

… like a group of local variables, a record, an array and any combination of the above …

How to read an entry in an array/record?

In its most basic form the value of a register is interpreted as an address and the memory content there is loaded into another register.
**ARM v7-M Copy data in and out of the CPU**

Most copy operations between CPU and memory follow this basic scheme.
ARM v7-M Move data in and out of the CPU

- **ldr**<c><q> <Rd>, [<Rb> {, #+/−<offset>}]
- **str**<c><q> <Rs>, [<Rb> {, #+/−<offset>}]

Reads from a potentially offset memory cell with a base register address.
ARM v7-M Move data in and out of the CPU

\[
\text{ldr}\langle c\rangle\langle q\rangle \ <Rd>, \ [\langle Rb\rangle \ {, \ #+/\-\langle offset\rangle}] \\
\text{str}\langle c\rangle\langle q\rangle \ <Rs>, \ [\langle Rb\rangle \ {, \ #+/\-\langle offset\rangle}] \\
\]

Writes to a potentially offset memory cell with a base register address.
ARM v7-M Move data in and out of the CPU

Immediate addressing
("Pre-indexed")

\[\text{ldr}<c><q> \quad <Rd>, \quad [<Rb>, \quad #+/-=<offset>]!\]
\[\text{str}<c><q> \quad <Rs>, \quad [<Rb>, \quad #+/-=<offset>]!\]

Reads from an offset memory cell with a base register address and writes the offset address back into the original base register.
ARM v7-M Move data in and out of the CPU

Immediate addressing
(“Pre-indexed”)

\[
\text{ldr} <c><q> \ <Rd>, [<Rb>, #+/-=<offset>]!
\]

\[
\text{str} \ <c><q> \ <Rs>, [<Rb>, #+/-=<offset>]!
\]

Writes to an offset memory cell with a base register address and writes the offset address back into the original base register.
ARM v7-M Move data in and out of the CPU

Immediate addressing ("Post-indexed")

\[ \text{ldr}<c><q> \quad \text{<Rd>}, \quad [\text{<Rb>}], \quad #+/^-\text{<offset>} \]

\[ \text{str}<c><q> \quad \text{<Rs>}, \quad [\text{<Rb>}], \quad #+/^-\text{<offset>} \]

Reads from a memory cell with a base register address and writes the offset address back into the original base register.
ARM v7-M Move data in and out of the CPU

Immediate addressing (“Post-indexed”)

\[
\text{\texttt{ldr} <c><q> <Rd>, [<Rb>], #/+<-offset>}
\]
\[
\text{\texttt{str} <c><q> <Rs>, [<Rb>], #/+<-offset>}
\]

Writes to a memory cell with a base register address and writes the offset address back into the original base register.
ARM v7-M Move data in and out of the CPU

Reads from a memory cell with a base register address plus a potentially shifted index register.

\[
\text{ldr}\ r1, [r4, r3]
\]

\[
\text{ldr}\ r1, [r4, r3, LSL #2]
\]
ARM v7-M Move data in and out of the CPU

Index register addressing

\[ \text{ldr}<c><q> \quad <Rd>, \ [<Rb>, \ <Ri> \ {, \ LSL \ #<\text{shift}>}] \]
\[ \text{str}<c><q> \quad <Rs>, \ [<Rb>, \ <Ri> \ {, \ LSL \ #<\text{shift}>}] \]

Writes to a memory cell with a base register address plus a potentially shifted index register.
ARM v7-M Move data in and out of the CPU

Reads from a data area embedded into the code section.

\[
\text{ldr}<c><q> \quad \text{<Rd>, <label>}
\]

\[
\text{ldr}<c><q> \quad \text{<Rd>, [PC, #+/<-offset>]
}\]

Note there is no store version.
ARM v7-M Move data in and out of the CPU

Multiple registers (positive growing stack)

```
stmia<c><q> <Rs>{!}, <registers>
ldmdb<c><q> <Rs>{!}, <registers>
```

Stores multiple registers into sequential memory addresses.
Stores “increment after” and loads “decrement before”.

© 2021 Uwe R. Zimmer, The Australian National University
ARM v7-M Move data in and out of the CPU

stmia<c><q>  <Rs>{!},  <registers>
ldmdb<c><q>  <Rs>{!},  <registers>

Reads multiple registers from sequential memory addresses.
Stores “increment after” and loads “decrement before”.

Note that any register can be use as stack base, i.e. you can have multiple stacks simultaneously.
ARM v7-M Move data in and out of the CPU

Stores multiple registers to sequential memory addresses.
Stores “decrement before” and loads “increment after”.

```asm
stmdb <c><q> <Rs>{!}, <registers>
ldmia <c><q> <Rs>{!}, <registers>
```

stmdb SP!, {r1, r3, r4, fp}
ARM v7-M Move data in and out of the CPU

stmdb <c><q> <Rs>{!}, <registers>
ldmia <c><q> <Rs>{!}, <registers>

Reads multiple registers from sequential memory addresses. Stores “decrement before” and loads “increment after”.

Multiple registers (negative growing stack)
Simple arithmetic in memory

Calculate again:

\[ e := a + b - 2c \]

but now \(a\), \(b\), \(c\) and \(e\) are stored in memory, relative to an address stored in FP ("Frame Pointer"): \(a\) is held at \([fp - 12]\), \(b\) at \([fp - 16]\), \(c\) at \([fp - 20]\) and \(e\) at \([fp - 24]\)

In order to do arithmetic we need to load those values into the CPU first and afterwards we need to store the result in memory:

\[
\begin{align*}
\text{ldr} & \quad r1, [fp, #-12] \\
\text{ldr} & \quad r2, [fp, #-16] \\
\text{add} & \quad r1, r1, r2 \\
\text{ldr} & \quad r2, [fp, #-20] \\
\text{lsl} & \quad r2, r2, #1 \\
\text{sub} & \quad r1, r1, r2 \\
\text{str} & \quad r1, [fp, #-24]
\end{align*}
\]

Notice that this time we only used two registers.
Simple arithmetic in memory

Calculate again:
\[
e := a + b - 2c
\]

Or in saturation arithmetic:
```
ldr r1, [fp, #-12]
ldr r2, [fp, #-16]
qadd r1, r1, r2
ldr r2, [fp, #-20]
qadd r2, r2, r2
qsub r1, r1, r2
str r1, [fp, #-24]
```
Calculate again:

\[ e := a + b - 2c \]

Or with overflow checks:

```
ldr r1, [fp, #-12]
ldr r2, [fp, #-16]
adds r1, r1, r2 ; need to check overflow flag
ldr r2, [fp, #-20]
lsr r2, r2, #1 ; need to check that the sign did not change
subs r1, r1, r2 ; need to check overflow flag
str r1, [fp, #-24]
```

☞ It’s time we learn about branching off into alternative execution paths.
**ARM v7-M Branch instructions**

- **b<q> label:** 
  \[ ; \text{if } c \text{ then } PC := \text{label} \]

- **bl<q> label:** 
  \[ ; \text{if } c \text{ then } LR := PC_{\text{next}}; PC := \text{label} \]

- **bx<Rm>:** 
  \[ ; \text{if } c \text{ then } PC := Rm \]

- **blx<c>q> <Rm>:** 
  \[ ; \text{if } c \text{ then } LR := PC_{\text{next}}; PC := Rm \]

- **cbz<q> Rn, label:** 
  \[ ; \text{if } Rn = 0 \text{ then } PC := \text{label} \]

- **cbnz<q> Rn, label:** 
  \[ ; \text{if } Rn \neq 0 \text{ then } PC := \text{label} \]

<table>
<thead>
<tr>
<th>&lt;c&gt;</th>
<th>Meanings</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>Equal</td>
<td>Z = 1</td>
</tr>
<tr>
<td>ne</td>
<td>Not equal</td>
<td>Z = 0</td>
</tr>
<tr>
<td>cs, hs</td>
<td>Carry set, Unsigned higher or same</td>
<td>C = 1</td>
</tr>
<tr>
<td>cc, lo</td>
<td>Carry clear, Unsigned lower</td>
<td>C = 0</td>
</tr>
<tr>
<td>mi</td>
<td>Minus, Negative</td>
<td>N = 1</td>
</tr>
<tr>
<td>pl</td>
<td>Plus, Positive or zero</td>
<td>N = 0</td>
</tr>
<tr>
<td>vs</td>
<td>Overflow</td>
<td>V = 1</td>
</tr>
<tr>
<td>vc</td>
<td>No overflow</td>
<td>V = 0</td>
</tr>
<tr>
<td>hi</td>
<td>Unsigned higher</td>
<td>C = 1 ∧ Z = 0</td>
</tr>
<tr>
<td>ls</td>
<td>Unsigned lower or same</td>
<td>C = 0 ∨ Z = 1</td>
</tr>
<tr>
<td>ge</td>
<td>Signed greater or equal</td>
<td>N = Z</td>
</tr>
<tr>
<td>lt</td>
<td>Signed less</td>
<td>N ≠ Z</td>
</tr>
<tr>
<td>gt</td>
<td>Signed greater</td>
<td>Z = 0 ∧ N = V</td>
</tr>
<tr>
<td>le</td>
<td>Signed less or equal</td>
<td>Z = 1 ∨ N ≠ V</td>
</tr>
<tr>
<td>al, &lt;none&gt;</td>
<td>Always</td>
<td>any</td>
</tr>
</tbody>
</table>
Calculate again:

e := a + b - 2*c

Or with overflow checks:

```assembly
ldr r1, [fp, #-12]
ldr r2, [fp, #-16]
adds r1, r1, r2
bvs Overflow ; branch if overflow is set
ldr r2, [fp, #-20]
adds r2, r2, r2
bvs Overflow ; branch if overflow is set
subs r1, r1, r2
bvs Overflow ; branch if overflow is set
str r1, [fp, #-24]
```

... 

**Overflow:**

```assembly
svc #5 ; call the operating system or runtime environment with #5 (assuming that #5 indicates an overflow situation)
```
Simple arithmetic in memory

Calculate again:

\[ e := a + b - 2c \]

Or with overflow checks:

\[
\begin{align*}
\textit{ldr} & \quad r1, [fp, #-12] \\
\textit{ldr} & \quad r2, [fp, #-16] \\
\textit{adds} & \quad r1, r1, r2 \\
\textit{bvs} & \quad \text{Overflow} \quad ; \text{branch if overflow is set} \\
\textit{ldr} & \quad r2, [fp, #-20] \\
\textit{adds} & \quad r2, r2, r2 \\
\textit{bvs} & \quad \text{Overflow} \quad ; \text{branch if overflow is set} \\
\textit{subs} & \quad r1, r1, r2 \\
\textit{bvs} & \quad \text{Overflow} \quad ; \text{branch if overflow is set} \\
\textit{str} & \quad r1, [fp, #-24] \\
\end{align*}
\]

... but how do we know where this happened or how to continue operations?

\[ \text{Overflow:} \]

\[
\begin{align*}
\textit{svc} & \quad #5 \quad ; \text{call the operating system or runtime environment with #5} \\
& \quad ; \text{(assuming that #5 indicates an overflow situation)}
\end{align*}
\]
Simple arithmetic in memory

Calculate again:

\[ e := a + b - 2c \]

Or with overflow checks:

```asm
    ldr r1, [fp, #-12]
    ldr r2, [fp, #-16]
    adds r1, r1, r2
    blvs Overflow ; branch if overflow is set; keep next location in LR
    ldr r2, [fp, #-20]
    adds r2, r2, r2
    blvs Overflow ; branch if overflow is set; keep next location in LR
    subs r1, r1, r2
    blvs Overflow ; branch if overflow is set; keep next location in LR
    str r1, [fp, #-24]
```

...  

"Overflow:"

```
    ... ; ... for example writing a log entry with location
    bx lr ; resume operations - assuming the above did not change LR
```
ARM v7-M Essential multiplications and divisions

32 bit to 32 bit

\[
\begin{align*}
\text{mul}\{s\}<c><q> \{<Rd>,\} <Rn>,<Rm> & \quad ; \quad Rd := \quad (Rn*Rm) \\
\text{mla}<c> \quad <Rd>, \quad <Rn>,<Rm>,<Ra> & \quad ; \quad Rd := Ra + (Rn*Rm) \\
\text{mls}<c> \quad <Rd>, \quad <Rn>,<Rm>,<Ra> & \quad ; \quad Rd := Ra - (Rn*Rm) \\
\text{udiv}<c> \quad <Rd>, \quad <Rn>,<Rm> & \quad ; \quad Rd := \text{unsigned} \ (Rn/Rm); \text{ rounded towards 0} \\
\text{sdiv}<c> \quad <Rd>, \quad <Rn>,<Rm> & \quad ; \quad Rd := \text{signed} \ (Rn/Rm); \text{ rounded towards 0}
\end{align*}
\]

32 bit to 64 bit

\[
\begin{align*}
\text{umull}<c> \quad <RdLo>,<RdHi>,<Rn>,<Rm> & \quad ; \quad RdHi:RdLo := \text{unsigned} \ (RdHi:RdLo + (Rn*Rm)) \\
\text{umlal}<c><q> \quad <RdLo>,<RdHi>,<Rn>,<Rm> & \quad ; \quad RdHi:RdLo := \text{unsigned} \ (RdHi:RdLo + (Rn*Rm)) \\
\text{smull}<c> \quad <RdLo>,<RdHi>,<Rn>,<Rm> & \quad ; \quad RdHi:RdLo := \text{signed} \ (RdHi:RdLo + (Rn*Rm)) \\
\text{smlal}<c> \quad <RdLo>,<RdHi>,<Rn>,<Rm> & \quad ; \quad RdHi:RdLo := \text{signed} \ (RdHi:RdLo + (Rn*Rm))
\end{align*}
\]

... versions for narrower numbers, as well as versions which operate on multiple narrower numbers in parallel exist as well.
Straight power

Calculate:

\[ c := a ^ b \]

\[ \begin{align*}
\text{mov} & \quad r1, \#7 \quad ; \ a \\
\text{mov} & \quad r2, \#11 \quad ; \ b \quad ; \text{has to be non-negative} \\
\text{mov} & \quad r3, \#1 \quad ; \ c \\
\text{power:} & \\
\text{cbz} & \quad r2, \text{end_power} \quad ; \text{exponent zero?} \\
\text{mul} & \quad r3, r1 \\
\text{sub} & \quad r2, \#1 \\
\text{b} & \quad \text{power} \\
\text{end_power:} & \\
\text{nop} & \quad ; \ c = a ^ b
\end{align*} \]

7^{11} = 7 \cdot 7 \cdot 7 \cdot 7 \cdot 7 \cdot 7 \cdot 7 \cdot 7 \cdot 7 \cdot 7

How many iterations?
How many cycles?
More power

Calculate:

\[ c := a^b \]

```
mov r1, #7 ; a
mov r2, #11 ; b ; has to be non-negative
mov r3, #1 ; c
mov r4, r1 ; base a to the powers of two, starting with \( a \cdot 1 \)

power:
  cbz r2, end_power ; exponent zero?
  tst r2, #0b1 ; right-most bit of exponent set?
  beq skip ; skip this power if not
  mul r3, r4 ; multiply the current power into result

skip:
  mul r4, r4 ; calculate next power
  lsr r2, #1 ; divide exponent by 2
  b power

end_power:
  nop ; c = a^b
```

\[ 7^{11} = 7^8 \cdot 7^2 \cdot 7^1 \]

How many iterations?
How many cycles?
Table based branching

\[ \text{tbb} \langle c \rangle \langle q \rangle \ [\langle Rn \rangle, \langle Rm \rangle] \quad ; \text{for tables of offset bytes (8bit)} \]
\[ \text{tbh} \langle c \rangle \langle q \rangle \ [\langle Rn \rangle, \langle Rm \rangle, \text{ls} \text{l} \ #1] \quad ; \text{for tables of offset halfwords (16bit)} \]

Common usage for byte (8bit) tables

\[ \text{tbb} \ [\text{PC, Ri}] \quad ; \text{PC is base of branch table, Ri is index} \]

**Branch_Table:**

\[ \text{.byte} \ (\text{Case}_A - \text{Branch_Table})/2 \ ; \text{Case}_A \ 8 \text{ bit offset} \]
\[ \text{.byte} \ (\text{Case}_B - \text{Branch_Table})/2 \ ; \text{Case}_B \ 8 \text{ bit offset} \]
\[ \text{.byte} \ (\text{Case}_C - \text{Branch_Table})/2 \ ; \text{Case}_C \ 8 \text{ bit offset} \]
\[ \text{.byte} \ 0x00 \quad ; \text{Padding to re-align with halfword boundaries} \]

**Case_A:**

\[ \ldots \quad ; \text{any instruction sequence} \]
\[ \text{b} \ \text{End}_Case \quad ; \text{“break out”} \]

**Case_B:**

\[ \ldots \quad ; \text{any instruction sequence} \]
\[ \text{b} \ \text{End}_Case \quad ; \text{“break out”} \]

**Case_C:**

\[ \ldots \quad ; \text{any instruction sequence} \]

**End_Case:**
Table based branching

```
tbb<c><q> [<Rn>, <Rm>] ; for tables of offset bytes (8bit)
tbh<c><q> [<Rn>, <Rm>, lsl #1]; for tables of offset halfwords (16bit)
```

Common usage for halfword (16bit) tables

```
tbh    [PC, Ri, lsl #1] ; PC used as base of branch table, Ri is index

Branch_Table:
  .hword (Case_A - Branch_Table)/2 ; Case_A 16 bit offset
  .hword (Case_B - Branch_Table)/2 ; Case_B 16 bit offset
  .hword (Case_C - Branch_Table)/2 ; Case_C 16 bit offset

Case_A:
  ...
  b End_Case ; any instruction sequence ; “break out”

Case_B:
  ...
  b End_Case ; any instruction sequence ; “break out”

Case_C:
  ...

End_Case:
```
### Basic instruction sets

<table>
<thead>
<tr>
<th>Category</th>
<th>Side effects</th>
<th>ARM v7-M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic, Logic</td>
<td>Sets and uses CPU flags</td>
<td><code>add, adc, qadd, sub, sbc, qsub, rsb, mul, mla, mls, udiv, sdiv, umull, umf, smul, smla, and, bic, orr, orn, eor, cmp, cmn, tst, teq</code></td>
</tr>
<tr>
<td>Move and shift registers</td>
<td></td>
<td><code>mov, lsr, asr, lsl, ror, rrx</code></td>
</tr>
<tr>
<td>Branching</td>
<td>Uses CPU flags</td>
<td><code>b, bl, bx, blx, tbb, tbh</code></td>
</tr>
<tr>
<td>Load &amp; Store</td>
<td>Effects memory</td>
<td><code>ldr, str, ldmdb, ldma, stmia, stmb</code></td>
</tr>
</tbody>
</table>
### Basic instruction sets

<table>
<thead>
<tr>
<th>Category</th>
<th>Side effects</th>
<th>ARM v7-M</th>
</tr>
</thead>
</table>
| Arithmetic, Logic       | Sets and uses CPU flags                                                       | add, adc, qadd, sub, sbc, qsub, rsb, 
|                         |                                                                               | mul, mla, mls, udiv, sdiv, umull, umlal, smull, smlal, 
|                         |                                                                               | and, bic, orr, orn, eor, cmp, cmn, tst, teq |
| Move and shift registers|                                                                               | mov, lsr, asr, lsl, ror, rrx       |
| Branching               | Uses CPU flags                                                               | b, bl, bx, blx, tbb, tbh           |
| Load & Store            | Effects memory                                                               | ldr, str, ldmdb, ldmia, stmia, stmdb|

Instruction sets in the field:

**RISC:** Power, ARM, MIPS, Alpha, SPARK, AVR, PIC, …

**CISC:** x86, Z80, 6502, 68000, …

Over 50 billion CPUs on this planet are running ARM instruction sets.
## Basic instruction sets

<table>
<thead>
<tr>
<th>Category</th>
<th>Side effects</th>
<th>ARM v7-M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic, Logic</td>
<td>Sets and uses CPU flags</td>
<td>add, adc, qadd, sub, sbc, qsub, rsb, mul, mla, mls, udiv, sdiv, umull, umlal, smull, smlal, and, bic, orr, orn, eor, cmp, cmn, tst, teq</td>
</tr>
<tr>
<td>Move and shift registers</td>
<td></td>
<td>mov, lsr, asr, lsl, ror, rrx</td>
</tr>
<tr>
<td>Branching</td>
<td>Uses CPU flags</td>
<td>b, bl, bx, blx, tbb, tbh</td>
</tr>
<tr>
<td>Load &amp; Store</td>
<td>Effects memory</td>
<td>ldr, str, ldmdb, ldmia, stmia, stmdb</td>
</tr>
</tbody>
</table>

What’s missing?

- Changing CPU privileges and handling interrupts.
- Synchronizing instructions

Coming in later chapters about concurrency and operating systems.
Summary

Hardware/Software Interface

- Instruction formats
  - Register sets
  - Instruction encoding

- Arithmetic / Logic instructions inside the CPU
  - Summation, Subtraction, Multiplication, Division
  - Logic and shift operations

- Load / Store and addressing modes
  - Direct, relative, indexed, and auto-index-increment addressing forms

- Branching
  - Conditional branching and unconditional jumps.