Lecture 6: More on assembly language programming

- The basic branch instruction is:
  
  B label ; unconditionally branch to label
  
  label .......

- Conditional branch instructions can be used to control loops:

  MOV r0, #10 ; initialize loop counted r0
  loop ........ ; start of body of loop
  SUB r0, r0, #1 ; decrement loop counter
  CMP r0, #0 ; is it zero yet?
  BNE loop ; branch if r0 ≠ 0

- The CMP instruction gives no results EXCEPT possibly changing conditional codes in CPSR.

  If r0=0, then Z bit is set (=1), else Z bit is reset (=0)

- The S-bit

  The loop program can be simplified to:

  MOV r0, #10 ; initialize loop counted r0
  loop ........ ; start of body of loop
  SUBS r0, r0, #1 ; decrement loop counter
  BNE loop ; branch if r0 ≠ 0

- Conditional Branch Instructions

- Conditional Execution

  Conditional execution applies not only to branches, but to all ARM instructions.

  For example:

<table>
<thead>
<tr>
<th>Branch</th>
<th>Interpretation</th>
<th>Normal uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Unconditional</td>
<td>Always take this branch</td>
</tr>
<tr>
<td>BNE</td>
<td>Not equal</td>
<td>Comparison not equal or non-zero result</td>
</tr>
<tr>
<td>CMP</td>
<td>Equal</td>
<td>Comparison equal or zero result</td>
</tr>
<tr>
<td>BMI</td>
<td>Minus</td>
<td>Result minus or negative</td>
</tr>
<tr>
<td>BCC</td>
<td>Carry clear</td>
<td>Arithmetic operation did not give carry-out</td>
</tr>
<tr>
<td>BLS</td>
<td>Carry set</td>
<td>Arithmetic operation gave carry-out</td>
</tr>
<tr>
<td>BHS</td>
<td>Higher or same</td>
<td>Signed comparison gave higher or same</td>
</tr>
<tr>
<td>BVC</td>
<td>Overflow clear</td>
<td>Signed integer operation; no overflow occurred</td>
</tr>
<tr>
<td>BVS</td>
<td>Overflow set</td>
<td>Signed integer operation; overflow occurred</td>
</tr>
<tr>
<td>BHI</td>
<td>Greater than</td>
<td>Signed integer comparison gave greater than</td>
</tr>
<tr>
<td>BGE</td>
<td>Greater or equal</td>
<td>Signed integer comparison gave greater or equal</td>
</tr>
<tr>
<td>BLT</td>
<td>Less than</td>
<td>Signed integer comparison gave less than</td>
</tr>
<tr>
<td>BLE</td>
<td>Less or equal</td>
<td>Signed integer comparison gave less than or equal</td>
</tr>
<tr>
<td>BHI</td>
<td>Higher</td>
<td>Unsigned comparison gave higher</td>
</tr>
<tr>
<td>BHS</td>
<td>Higher or same</td>
<td>Signed comparison gave higher or same</td>
</tr>
<tr>
<td>BLS</td>
<td>Lower or same</td>
<td>Unsigned comparison gave lower or same</td>
</tr>
</tbody>
</table>

  Note that BCC = BLO, BCS = BHS

  Can be replaced by:

  CMP r0, #5 ; if (r0 != 5) then
  BEQ BYPASS
  ADD r1, r1, r0 ; r1 := r1 + r0 - r2
  SUB r1, r1, r2
  BYPASS ......

  Here the ADDNE and SUBNE instructions are executed only if Z='0', i.e. the CMP instruction gives non-zero result.
Conditional Execution - more

- Here is another very clever use of this unique feature in ARM instruction set. Do remember ALL instructions can be qualified by the condition codes.

```c
; if ( (a==b) && (c==d)) then e := e + 1;
CMP r0, r1 ; r0 has a, r1 has b
CMPEQ r2, r3 ; r2 has c, r3 has d
ADDEQ r4, r4, #1 ; e := e+1
```

- Note how if the first comparison finds unequal operands, the second and third instructions are both skipped.
- Also the logical 'and' in the if clause is implemented by making the second comparison conditional.
- Conditional execution is only efficient if the conditional sequence is three instructions or fewer. If the conditional sequence is longer, use a proper loop.

Conditional Execution - Summary

<table>
<thead>
<tr>
<th>Numeric</th>
<th>Condition</th>
<th>CPU condition flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>CEqual</td>
<td>Z set</td>
</tr>
<tr>
<td>NE</td>
<td>Not Equal</td>
<td>Z clear</td>
</tr>
<tr>
<td>CS</td>
<td>CmSetFused Higher Or Same</td>
<td>C set</td>
</tr>
<tr>
<td>CC</td>
<td>CcSetFused Higher Or Same</td>
<td>C clear</td>
</tr>
<tr>
<td>MI</td>
<td>Negative (Minus)</td>
<td>N set</td>
</tr>
<tr>
<td>PL</td>
<td>Positive (Plus)</td>
<td>N clear</td>
</tr>
<tr>
<td>VS</td>
<td>oVerflow Set</td>
<td>V set</td>
</tr>
<tr>
<td>VC</td>
<td>oVerflow Clear</td>
<td>V clear</td>
</tr>
<tr>
<td>HI</td>
<td>Higher Unsigned</td>
<td>C set and Z clear</td>
</tr>
<tr>
<td>LS</td>
<td>Lower or Signed</td>
<td>C clear or Z set</td>
</tr>
<tr>
<td>GE</td>
<td>Greater than or Equal to</td>
<td>(N and Vt set or (NAnd Vt) clear</td>
</tr>
<tr>
<td>LT</td>
<td>Less than</td>
<td>(N set and Vt set) or ((N and Vt) set and Z clear</td>
</tr>
<tr>
<td>GT</td>
<td>Greater Than</td>
<td>(N set and Vt set) or (NAnd Vt) set</td>
</tr>
<tr>
<td>LE</td>
<td>Less Than or Equal to</td>
<td>(N set and Vt set) or (NAnd Vt) set or Z set</td>
</tr>
</tbody>
</table>

Shifted Register Operands

- ARM has another very clever feature. In any data processing instructions, the second register operand can have a shift operation applied to it. For example:

```c
ADD  r3, r2, r1, LSL #3 ; r3 := r2 + 8 x r1
```

- Here LSL means 'logical shift left by the specified number of bits.
- Note that this is still a single ARM instruction, executed in a single clock cycle.
- In most processors, this is a separate instructions, while ARM integrates this shifting into the ALU.
- It is also possible to use a register value to specify the number of bits the second operand should be shifted by:

```c
ADD  r5, r5, r3, LSL r2 ; r5 := r5 + r3 x 2**r2
```

ARM shift operations - LSL and LSR

- Here are all the six possible ARM shift operations you can use:

```
<table>
<thead>
<tr>
<th>Numeric</th>
<th>Condition</th>
<th>CPU condition flags</th>
</tr>
</thead>
</table>
```

- LSL: logical shift left by 0 to 31 places; fill the vacated bits at the least significant end of the word with zeros.
- LSR: logical shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros.
ARM shift operations - ASL and ASR

- ASL: arithmetic shift left; this is the same as LSL
- ASR: arithmetic shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros if the source operand was positive, and with ones if it is negative. That is, sign extend while shifting right.

![ASL and ASR diagrams]

ARM shift operations - ROR and RRX

- ROR: rotate right by 0 to 32 places; the bits which fall off the least significant end are used to fill the vacated bits at the most significant end of the word.
- RRX: rotate right extended by 1 place; the vacated bit (bit 31) is filled with the old value of the C flag and the operand is shifted one place to the right. This is effectively a 33 bit rotate using the register and the C flag.

A simple assembly language program - Hello world!

- We will now consider two simple assembly language programs. The first outputs "Hello World!" on the console window:

```assembly
AREA helloW, CODE, READONLY
ENTRY
START
LDRB r0, [r1], #1
CMP r0, #0
SWINE SWI_WriteC
BNE LOOP
SWI SWI_Exit
TEXT = "Hello World!", &0a, &0d, 0
END
```

```assembly
AREA helloW, CODE, READONLY
ENTRY
START
LDR r0, [r1], #4
STR r0, [r2], #4
CMP r0, #0
SWINE SWI_WriteC
BNE LOOP2
SWI SWI_Exit
TABLE1 = "This is the right string!", &0a, &0d, 0
T1END
ALIGN
TABLE2 = "This is the wrong string!", 0
END
```

Another Example: Block copy

- Here is another example to block copy from one address (TABLE1) to another (TABLE2), then write it out:

```assembly
AREA BlkCpy, CODE, READONLY
ENTRY
START
ADR r1, TABLE1
ADR r2, TABLE2
ADR r3, T1END
LOOP1 LDR r0, [r1], #4
STR r0, [r2], #4
CMP r1, r3
BLT LOOP1
ADR r1, TABLE2
LOOP2 CMP r0, #0
SWINE SWI_WriteC
BNE LOOP2
SWI SWI_Exit
TABLE1 = "This is the right string!", &0a, &0d, 0
T1END
ALIGN
TABLE2 = "This is the wrong string!", 0
END
```