Lecture 5 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           ; intialize 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 ≠ 0c
```

- 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 simplied to:

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

- SUBS instruction is the same as SUB except that the former can change the cc in CPSR.
- After SUBS instruction, Z-bit is set or cleared depending on the result of the subtraction.
- All data processing instructions such as ADD, ADC etc. can have S added to indicate that condition code should be updated.
- We have seen how to use a conditional branch instruction BNE in loops. There are in fact many more conditional branch instructions.

Conditional Branch Instructions

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

- Note that BCC = BLO, BCS = BHS

Conditional Execution

- Conditional execution applies not only to branches, but to all ARM instructions.
- For example:

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

- Can be replaced by:

```
        CMP     r0, #5            ; if (r0 != 5) then
        ADDNE   r1, r1, r0        ;     r1 := r1 + r0 - r2
        SUBNE   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.

```
;  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 if 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

Mnemonic	Condition	CPU condition flags
EQ	EQual	Z set
NE	Not Equal	Z clear
CS	Carry Set/unsigned Higher or Same	C set
CC	Carry Clear/unsigned LOwer than	C clear
MI	Negative (MInus)	N set
PL	Positive (PLus)	N clear
VS	oVerflow Set	V set
VC	oVerflow Clear	V clear
HI	HIgher unsigned	C set and Z clear
LS	Lower or Same unsigned	C clear or Z set
GE	Greater than or Equal to	(N and V) set or (Nand V) clear
LT	Less Than	(N set and V clear) or (N clear and V set)
GT	Greater Than	((N and V) set or clear) and Z clear
LE	Less Than or equal to	(N set and V clear) or (N clear and V set) or Z set

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:

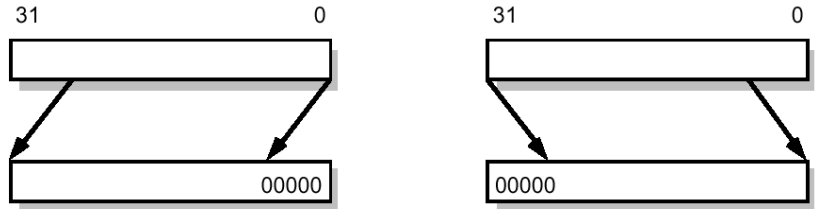
```
ADD      r3, r2, r1, LSL #3          ; r3 := r2 + 8 x r1
```

- Herer 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 cyle.
- 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:

```
ADD      r5, r5, r3, LSL r2          ; r5 := r5 + r3 x 2**r1
```


ARM shift operations - LSL and LSR

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

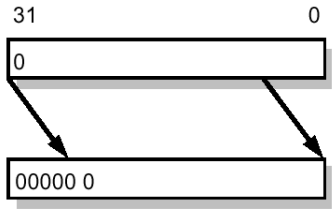


LSL #5 LSR #5

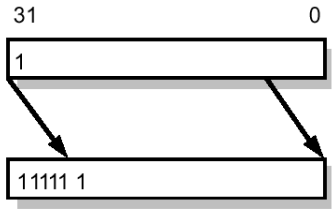
- LSL: logical shift left by 0 to 31 places; fill the vacated bits at the least significant end of the word with zeros.
- LSR: logical shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros.

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 it is negative. That is, sign extend while shifting right.



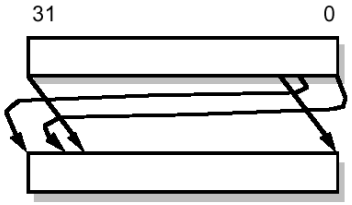
`ASR #5`, positive operand



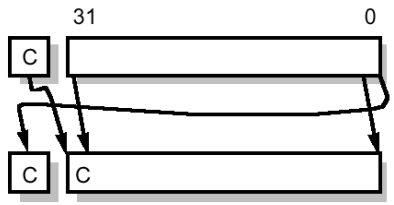
`ASR #5`, negative operand

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.



`ROR #5`



`RRX`

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:

```
            AREA    helloW, CODE, READONLY  ; declare code area
SWI_WriteC  EQU     &0                      ; output character in r0
SWI_Exit    EQU     &11                     ; finish program
            ENTRY                           ; code entry point
START       ADR     r1, TEXT                ; r1 -> "Hello World!"
LOOP        LDRB    r0, [r1], #1            ; get the next byte
            CMP     r0, #0                  ; check for 'null' character
            SWINE   SWI_WriteC              ; if not end, print ....
            BNE     LOOP                    ;  ... and loop back
            SWI     SWI_Exit                ; end of execution

TEXT        =       "Hello World!", &0a, &0d, 0  ; string + CR + LF + null
            END
```


Another Example: Block copy

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

```
            AREA    BlkCpy, CODE, READONLY  ; declare code area
SWI_WriteC  EQU     &0                      ; output character in r0
SWI_Exit    EQU     &11                     ; finish program
            ENTRY                           ; code entry point
START       ADR     r1, TABLE1              ; r1 -> TABLE1
            ADR     r2, TABLE2              ; r2 -> TABLE2
            ADR     r3, T1END               ; r3 -> end of TABLE1
LOOP1       LDR     r0, [r1], #4            ; get TABLE1 1st word
            STR     r0, [r2], #4            ; copy into TABLE2
            CMP     r1, r3                  ; finished?
            BLT     LOOP1                   ; if not, do more, else print
            ADR     r1, TABLE2              ; r1 -> TABLE2
LOOP2       CMP     r0, #0                  ; check for end of text string
            SWINE   SWI_WriteC              ; if not end, print ...
            BNE     LOOP2                   ; .... and loop back
            SWI     SWI_Exit                ; finish
TABLE1      =       "This is the right string!", &0a, &0d, 0
T1END
            ALIGN                           ; ensure word alignment
TABLE2      =       "This is the wrong string!", 0
            END
```