Control Flow and Loops

Steven R. Bagley

Introduction

- Started to look at writing ARM Assembly Language
- Saw the structure of various commands
- Load (`LDR`), Store (`STR`) for accessing memory
- `SWI`s for OS access
- Data Processing Instructions
- Assembler Directives

```
        B main

num     DEFW  4
success DEFB  "R0 has reached the value of \0"

        ALIGN

main    LDR   R1, num
        MOV   R0, #1
next    CMP   R0, R1
        BNE   skip
        ADR   R0, success
        SWI   3
        MOV   R0,R1
        SWI   4
        MOV   R0, #10
        SWI   0
        SWI   2

skip    ADD   R0, R0, #1
        B     next
```

Highlight Conditional branches follows the CMP

Conditional Branch

- Saw Branch (B) instructions earlier
- These branches have extra letters in the mnemonics (BNE)
- These are *conditional* branches, branches only if the condition is true
- What condition?

Condition Codes

	Interpretation	CCs for execution
EQ	Equal / Equals Zero	Z
NE	Not Equal	$\overline{Z}$
CS/HS	Carry Set / Higher or same (unsigned)	C
CC/LO	Carry Clear / Lower (unsigned)	$\overline{C}$
MI	Minus / Negative	N
PL	Plus / Positive or zero	$\overline{N}$
VS	Overflow Set	$C \cdot \overline{Z}$
VC	Overflow Clear	$\overline{C} + Z$

Condition Codes

	Interpretation	CCs for execution
HI	Unsigned higher	$C.\overline{Z}$
LS	Unsigned lower or same	$\overline{C} + Z$
GE	Greater than or Equal (signed)	N = V
LT	Less than (signed)	N != V
GT	Greater than (signed)	$\overline{Z}$. (N = V)
LE	Less than or equal (signed)	Z + (N != V)
AL	Always	any
NV	Never (do not use!)	none

Conditional Branch

- Condition is based on the state of flags in a special register, the `CPSR`
- Current Program Status Register
- Not part of the normal 16 registers
- Certain instructions can set flags in the `CPSR` depending on the result of the calculation…
- Requires an `S` suffix to the instruction

Originally used unused bits in R15, but has since been separated and extended…

Most CPUs always update the flags, but the ARM makes the programmer do it manually

Data Processing Instructions

Mnemonic	Meaning	Effect

Data Processing Instructions

Mnemonic	Meaning	Effect
AND	Logical bit-wise AND	`Rd = Rn AND Op`
EOR	Logical bit-wise exclusive OR	`Rd = Rn EOR Op2`
SUB	Subtract	`Rd = Rn - Op2`
RSB	Reverse Subtract	`Rd = Op2 - Rn`
ADD	Add	`Rd = Rn + Op2`
ADC	Add with carry	`Rd = Rn + Op2 + C`
SBC	Subtract with Carry	`Rd = Rn - Op2 + C - 1`
RSC	Reverse Subtract with Carry	`Rd = Op2 - Rn + C - 1`

Data Processing Instructions

Mnemonic	Meaning	Effect

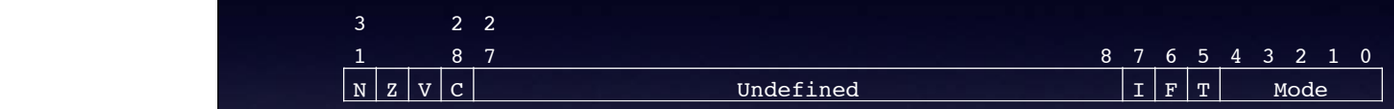
TST, TEQ, CMP and CMN have no destination

Data Processing Instructions

Mnemonic	Meaning	Effect
TST	Logical bit-wise AND	Sets condition on `Rn AND Op2`
TEQ	Logical bit-wise exclusive OR	Sets condition on `Rn EOR Op2`
CMP	Subtract	Sets condition on `Rn - Op2`
CMN	Compare negated	Sets condition on `Rn + Op2`
ORR	Logical Bit-wise OR	`Rd = Rn OR Op2`
MOV	Move	`Rd = Op2`
BIC	Bit clear	`Rd = Rn AND NOT Op2`
MVN	Move NOT	`Rd = NOT Op2`

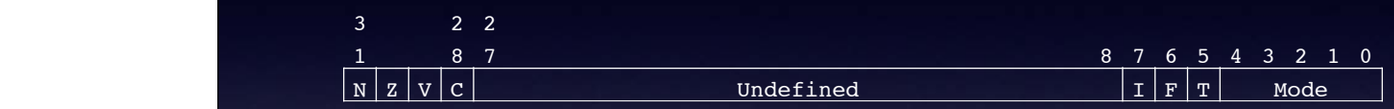
TST, TEQ, CMP and CMN have no destination

CPSR



Carry flag, last operation caused a carry to be set

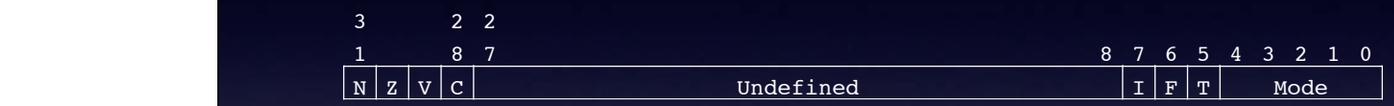
CPSR



N	Last operation yielded negative result

Carry flag, last operation caused a carry to be set

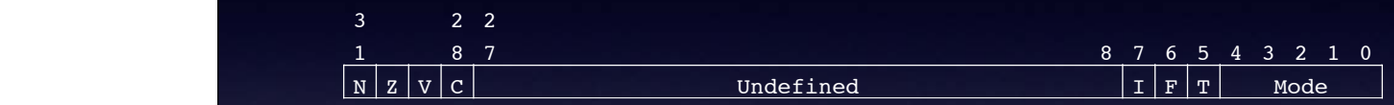
CPSR



N	Last operation yielded negative result
Z	Last operation yielded a zero result

Carry flag, last operation caused a carry to be set

CPSR



N	Last operation yielded negative result
Z	Last operation yielded a zero result
V	Overflow bit…

Carry flag, last operation caused a carry to be set

CPSR

N	Last operation yielded negative result
Z	Last operation yielded a zero result
V	Overflow bit…
C	Carry flag

Carry flag, last operation caused a carry to be set

Overflow Bit

- Shows that an invalid result was generated when adding signed numbers
- Adding two positive (or negative) numbers could give a result greater than 2^{31} or less than 2^{-31}
- This flips the sign of the numbers and produces an invalid result
- Adding a positive and a negative number will always work…

Read http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt and link on the website

Remember though that the CPU knows nothing about signed numbers, it's a trick of the encoding…

Demo on paper using two 8-bit numbers

4-bit signed encodings

Sign and Magnitude		Excess-3		One's Complement		Two's Complement	
1111	−7	0000	−3	1000	−7	1000	−8
1110	−6	0001	−2	1001	−6	1001	−7
1101	−5	0010	−1	1010	−5	1010	−6
1100	−4	0011	0	1011	−4	1011	−5
1011	−3	0100	+1	1100	−3	1100	−4
1010	−2	0101	+2	1101	−2	1101	−3
1001	−1	0110	+3	1110	−1	1110	−2
1000	−0	0111	+4	1111	−0	1111	−1
0000	+0	1000	+5	0000	+0	0000	0
0001	+1	1001	+6	0001	+1	0001	+1
0010	+2	1010	+7	0010	+2	0010	+2
0011	+3	1011	+8	0011	+3	0011	+3
0100	+4	1100	+9	0100	+4	0100	+4
0101	+5	1101	+10	0101	+5	0101	+5
0110	+6	1110	+11	0110	+6	0110	+6
0111	+7	1111	+12	0111	+7	0111	+7

Very similar to one's complement but we invert negative numbers and add one
Only one zero, but we have one more negative number than positive
Addition same as with unsigned numbers

CMP instruction

- Mnemonic: CMP
- Operands: 2 source registers (result not stored)
- Calculates `Rm` - `Rn` and updates status registers
- If registers are equal, `Z` flag is set
- So `BEQ` and `BNE` work as expect
- Also `CMN` which adds the operands…

CMN == Compare Negative

TST, and TEQ all follow the same structure

```
MOVS R0, #0
BEQ  foo      ; Branches because R0 contains zero

ADDS R0, R0, #1 ; Sets condition flags
ADD  R0, R0, #1 ; Does not set flags

MVNS R0, #0 ; R0 = -1
BMI foo     ; Jumps because R0 contains a -ve number
```

Should be able to understand all of this program now

```
        B main

num     DEFW  4
success DEFB  "R0 has reached the value of \0"

        ALIGN

main    LDR   R1, num
        MOV   R0, #1
next    CMP   R0, R1
        BNE   skip
        ADR   R0, success
        SWI   3
        MOV   R0,R1
        SWI   4
        MOV   R0, #10
        SWI   0
        SWI   2

skip    ADD   R0, R0, #1
        B     next
```

Should be able to understand all of this program now

Writing Assembly Programs

- One way to start writing assembly programs is to start by thinking about the C version
- And then converting it to assembler in stages
- First remove all the high-level features that C doesn't have
- Then start assign variables into registers
- But you may also need to use memory to store some variables too (can use `DEFW` to create space for this)

From C to ARM assembler

- How would we translate a single instruction from C to ARM?
- One C instructions can pack a lot of functionality
- Need to break it down into each of the individual components
- Lets look at an example

```
e = a + b - c * d;
```

It’s a simple C command, but it actually does eight things…

From C to Assembler

```
e = a + b - c * d;
```

- Assembly instructions generally only do one thing
- Need to break the C instruction down into those individual instructions
- Make sure arrange them in the right order…
- Let's try it with this example…

Switch to text editor and get students to start breaking it down (preload it with the DEFWs needed for a-e)

```
        B main

a       DEFW 4
b       DEFW 2
c       DEFW 8
d       DEFW 16
e       DEFW 0
        ALIGN

main    LDR r0, a
        LDR r1, b
        ADD r0, r0, r1
        LDR r1, c
        LDR r2, d
        MUL r3, r1,r2
        SUB r0, r0, r3
        STR r0, e
```

Could optimise this a bit though

High-level structures

- If we aren't careful our programs can become unreadable
- Often referred to as 'spaghetti code'
- Particularly if people make use of the `goto` command
- Hence we make use of control structures to structure our programs

Hence, 'goto"s outlawed nature and omission from many languages

Assembler Structure

- Unfortunately, Assembler has no high-level structures
- Only have the equivalent of `goto`, branch (`B`)
- This is how the hardware works
- *Unconditional* branches happen all the time
- *Conditional* branches only happen if some condition is met

Saw the different types last lecture…

C Compiler

- C Compiler regularly converts C to assembler
- Uses the same assembler structures to represent an `if` statement (etc.) every time
- Can use a similar approach ourselves in converting from C to assembler
- Can then optimise it later on…

Let's take a look at what the C compiler produces…

Works here because I have the iPhone SDK installed…

Obviously, as we get more experienced we'll end up writing the optimised assembler version straight away…

```
if(c == '1')
{
    printf("Have …\n”);
}
```

```
    cmp     r0, #49
    bne     LBB0_2
    b       LBB0_1
LBB0_1:
    ldr     r0, LCPI0_0
LPC0_0:
    add     r0, pc, r0
    bl      _printf
    str     r0, [sp]
@ 4-byte Spill
LBB0_2:
```

Could optimise this a lot though

Highlighted code is only executed when r0 = #49 otherwise it is skipped over (due to the bne)

Note that the compiler produces a lot of silly code in its default state — we would want to optimize this (and would never write it by hand)

```
if(c == '1')
{
    printf("Have …\n”);
}
```

```
    cmp     r0, #49
    bne     LBB0_2
    b       LBB0_1
LBB0_1:
    ldr     r0, LCPI0_0
LPC0_0:
    add     r0, pc, r0
    bl      _printf
    str     r0, [sp]
@ 4-byte Spill
LBB0_2:
```

Could optimise this a lot though

Highlighted code is only executed when r0 = #49 otherwise it is skipped over (due to the bne)

Note that the compiler produces a lot of silly code in its default state — we would want to optimize this (and would never write it by hand)

`ifs` in Assembler

- Do a `CMP` to test for the condition
- Branch if the condition is *not* met to skip over the code
- Code is then executed only if the condition is met…
- Otherwise we skip over it…

`if` Examples

C	Assembler

`if(R0 == 10)`
`{`
`  …`
`}`
	`  CMP R0, #10`
	`  BNE skip`
	`  …`
	`skip`

`if(R0 < R1)`
`{`
`  …`
`}`
	`  CMP R0, R1`
	`  BGE skip`
	`  …`
	`skip`

`if(R0 >= 10)`
`{`
`  …`
`}`
	`  CMP R0, #10`
	`  BLT skip`
	`  …`
	`skip`

`if(R0 != 0)`
`{`
`  …`
`}`
	`  CMP R0, #10`
	`  BEQ skip`
	`  …`
	`skip`

```
        B main

menu    DEFB “G51CSA Vending Machine…\n\0”
coke    DEFB “Have a bottle of Coke\n\0”
        ALIGN

main    SWI 1           ; Read a character
        CMP R0, #49
        BNE skip
        ADR R0, coke
        SWI 3
skip    SWI 2
```

Beginning of the Coke vending machine from G51PRG

label name for skip unimportant — but it needs to be unique

```
        B main

menu    DEFB “G51CSA Vending Machine…\n\0”
coke    DEFB “Have a bottle of Coke\n\0”
        ALIGN

main    SWI 1           ; Read a character
        CMP R0, #49
        BNE skip
        ADR R0, coke
        SWI 3
skip    SWI 2
```

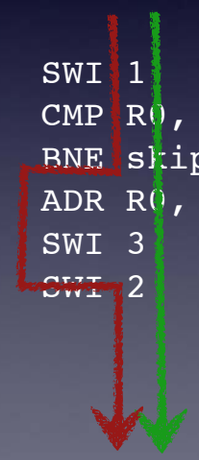
Beginning of the Coke vending machine from G51PRG

label name for skip unimportant — but it needs to be unique

```
        B main

menu    DEFB "G51CSA Vending Machine…\n\0"
coke    DEFB "Have a bottle of Coke\n\0"
        ALIGN

main    SWI 1           ; Read a character
        CMP R0, #49
        BNE skip
        ADR R0, coke
        SWI 3
skip    SWI 2
```



Beginning of the Coke vending machine from G51PRG

label name for skip unimportant — but it needs to be unique