Overview

- Compare instruction mechanism
- Flag setting Instructions
- Conditional Instructions
- Conclusion

Review (#1/2)

- HLL decisions (if, case) and loops (while, for) use same assembly instructions
  - Compare instruction: `cmp` in ARM
  - Conditional branches: `beq, bne, bgt, blt, etc` in ARM
  - Unconditional branches: `b, and mov pc, rn` in ARM
  - Switch/Case: chained if-else or jump table + `ldr pc, [ ]`

- `ldr pc, [ ]` is VERY POWERFUL!

Review (#2/2)

- Some Branch Conditions after `cmp` instruction:
  - `b` Unconditional
  - `bal` Branch Always
  - `beq` Branch Equal
  - `bne` Branch Not Equal
  - `blt` Branch Less Than
  - `ble` Branch Less Than or Equal
  - `bgt` Branch Greater Than
  - `bge` Branch Greater Than or Equal

Review: Branches: PC-relative addressing

- Recall register r15 in the machine also called PC;
- Points to the currently executing instruction
- Most instruction add 4 to it. (PC increments by 4 after execution of most instructions)
- Branch changes it to a specific value
- Branch adds to it
  - 24-bit signed value (contained in the instruction) r14
  - Shifted left by 2 bits
- Labels => addresses

Review: Status Flags in Program Status Register CPSR

Cmp Instructions and CPSR Flags (#1/3)

`cmp r1, r2 ; update flags after r1–r2`

---

Cmp Instructions and CPSR Flags (#2/3)

`cmp r1, r2 ; update flags after r1–r2`

---

Copies of the ALU status flags (latched for some instructions).

---

indicates a negative number in signed operations

Result of operation was zero.

Result was greater than 32 bits (for unsigned arithmetic, bit 31 is the magnitude bit, and not the sign bit).

Result was greater than 31 bits (for signed arithmetic bit 31 is the sign bit and not the magnitude bit).

Indicates a possible corruption of the sign bit in signed numbers (carry into the msb ≠ carry out of msb)
### cmp Instructions and CPSR Flags (#3/3)

```
cmp r1, r2
; Update flags after r1-r2
```

<table>
<thead>
<tr>
<th>31  28  24  20  16  12  8  4  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>N    Z   C    V</td>
</tr>
</tbody>
</table>

- $x1xx = Z$ set (equal) (eq)
- $1xx0 = N$ set and $V$ clear (signed less) (le)
- $0xx1 = N$ clear and $V$ set (signed less) (le)

### Example

**Assuming** $r1 = 0x7fffffff$, and $r2 = 0xffffffff$

What the CPSR flags would be after the instruction; `cmp r1, r2`

**Answer:**

- $N=1$, $Z=0$, $C=0$, $V=1$

**Explanation:**

1. Calculate $r1 - r2$: $0x7fffffff - 0xffffffff = 0x80000000$
2. The result is a negative number, so $N$ is set, and $C$ is not set.
3. The result is not zero, so $Z$ is not set.
4. The result is not equal to $r2$, so $V$ is set.

### Branch Instruction Conditional Execution Field

```
cmp r1, r2
B{<cont>} xyz
```

<table>
<thead>
<tr>
<th>31  28  24  20  16  12  8  4  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COND 101 1 24-bit signed offset</td>
</tr>
</tbody>
</table>

- $0000 = EQ$ - $Z$ set (equal)
- $0001 = NE$ - $Z$ clear (not equal)
- $0010 = HS$ / $CS$ - $C$ set (unsigned higher or same)
- $0011 = LO$ / $CC$ - $C$ clear (unsigned lower)
- $0100 = MI$ - $N$ set (negative)
- $0101 = PL$ - $N$ clear (positive or zero)
- $0110 = VS$ - $V$ clear (overflow)
- $0111 = VC$ - $V$ clear (no overflow)
- $1000 = HI$ - $C$ set and $Z$ clear (unsigned higher)

### Flag Setting Instructions

- **Compare**
  
  `cmp r1, r2 ; Update flags after r1-r2`

- **Compare Negated**
  
  `cmn r1, r2 ; Update flags after r1 + r2`

- **Test**
  
  `tst r1, r2 ; Update flags after r1 AND r2`

- **Test Equivalence**
  
  `teq r1, r2 ; Update flags after r1 EOR r2`

These instructions **DO NOT** save results; Only **UPDATE CPSR Flags**
Flag Setting Instructions Example

- Assuming \( r_1 = 0x7fffffff \), and \( r_2 = 0xffffffff \)

What the CPSR flags would be after the instructions

\[
\begin{align*}
\text{cmp } r_1, r_2, \text{ lsl } #2 \\
\text{cmn } r_1, r_2, \text{ lsl } #2 \\
\text{tst } r_1, r_2, \text{ lsr } #1 \\
\text{teq } r_1, r_2, \text{ lsr } #1
\end{align*}
\]

Flag Setting Instructions Example Solution (#1/4)

\[
\begin{align*}
\text{cmp } r_1, r_2, \text{ lsl } #2 \\
\text{Answer:} \\
r_1 = 0x7fffffff = 2147483647 \\
r_2 \text{ lsl } #2 = 0xfffffffc = 4294967292 \text{ (unsigned)} \\
&= (-4) \text{ (signed)} \\
r_1 - (r_2, \text{ lsl } #2) =
\end{align*}
\]

\[
\begin{align*}
0x7fffffff + 0x00000004 &\quad 2\text{'s complement of } 0xfffffffc \\
0x80000003 &\quad \text{(carry into msb = carry out of msb)} \\
0 &\quad \text{(carry out)} \\
\end{align*}
\]

\( N=1, Z=0, C=0, V=1 \)

C clear \( \rightarrow \) (unsigned lower) \( \text{lo/cc} \)

2147483647 < 4294967292

Z clear AND (\( N=V \)) \( \rightarrow \) (signed greater than) \( \text{gt} \)

2147483647 > -4

Flag Setting Instructions Example Solution (#2/4)

\[
\begin{align*}
\text{cmn } r_1, r_2, \text{ lsl } #2 \\
\text{Answer:} \\
r_1 = 0x7fffffff \\
r_2 \text{ lsl } #2 = 0xfffffffc = (-4) \text{ (signed)} \\
r_1 + r_2, \text{ lsl } #2 = r_1 - (-r_2 <<2) \text{ (comparing } r_1 \text{ and } -r_2) \\
0x7fffffff + 0xfffffffc \\
0x7fffffff \text{(carry into msb = carry out of msb)}
\end{align*}
\]

\( 1 \text{ (carry out)} \)

\( N=0, Z=0, C=1, V=0 \)

C set \( \rightarrow \) (unsigned higher or same) \( \text{hs/cc} \).

C set here really means there was an unsigned addition overflow

Z clear AND (\( N=V \)) \( \rightarrow \) (signed greater than) \( \text{gt} \)

2147483647 > -(-4) = 4

Flag Setting Instructions Example Solution (#3/4)

\[
\begin{align*}
\text{tst } r_1, r_2, \text{ lsr } #1 \\
\text{Answer:} \\
r_1 = 0x7fffffff \\
r_2, \text{ lsr } #1 = 0x7fffffff
\end{align*}
\]

\[
\begin{align*}
\text{lsr} \\
0 \quad \text{Destination} \\
C
\end{align*}
\]

\( r_1 \text{ and } r_2 \text{ lsr } #1 =
\]

\[
\begin{align*}
0x7fffffff + 0x00000000 &\quad \text{(carry into msb = carry out of msb)} \\
0x00000000 &\quad \text{(carry out)}
\end{align*}
\]

\( N=0, Z=0, C=1, V=0 \)

N clear \( \rightarrow \) (Positive or Zero) \( \text{pl} \)

Z clear \( \rightarrow \) (Not Equal) \( \text{ne} \). It really means ANDing of \( r_1 \) and \( r_2 \) does not make all bits 0 , ie \( r_1 \text{ AND } r_2 \neq 0 \)
Flag Setting Instructions Example Solution (#4/4)

teq r1, r2, lsr #1

Answer:
r1 = 0x7fffffff
R2, lsr #1 = 0x7fffffff

\[ r1 \text{ eor } r2 \text{ lsl } #1 = \]
\[ 0x7fff \, ffff \text{ xor } 0x7fff \, ffff \]
\[ 0x0000 \, 0000 \]

N, Z = 1, C = 1, V = 0

N clear \to (\text{Positive or Zero}) (pl)
Z set \to (\text{Equal}) \text{ It means } r1 = r2

Updating CPSR Flags with Data Processing Instructions

° By default, data processing operations do not affect the condition flags

- add r0, r1, r2 ; r0 = r1 + r2
  ; ... and DO not set ; flags

° To cause the condition flags to be updated, the “S” bit of the instruction needs to be set by postfixing the instruction with an “S”.

  • For example to add two numbers and set the condition flags:

- adds r0, r1, r2; r0 = r1 + r2
  ; ... and DO set flags

Updating Flags Example

° Compile this C code into ARM:

\[
\text{sum} = 0; \\
\text{for } (i=0; i<10; i=i+1) \\
\quad \text{sum} = \text{sum} + \text{A}[i]; \\
\quad \text{sum:}v1, v2, \text{base address of A:}v3
\]

\[
\text{mov v1, #0} \\
\text{mov v2, #0} \\
\text{A Loop: ldr al, [v3, v2, lsl #2] ; al=A[i]} \\
\quad \text{add v1, v1, al} \quad ; \text{sum = sum+A[i]} \\
\quad \text{add v2, v2, #1} \quad ; \text{increment i} \\
\quad \text{cmp v2, #10} \quad ; \text{Check(i<10)} \\
\quad \text{bne Loop} \quad ; \text{goto loop}
\]
Updating Flags Example Solution Ver 2

```
sum = 0;
for (i=0; i<10; i=i+1)
    sum = sum + A[i];
```

- sum: v1, i: v2, base address of A: v3

```
mov v1, #0
mov v2, #9 ; start with i = 9
loop: ldr al, [v3, v2, lsl #2] ; a1 = A[i]
    add v1, v1, a1 ; sum = sum + A[i]
    sub v2, v2, #1 ; decrement i
    cmp v2, #0 ; Check (i<0)
    bge loop ; goto loop
```

Updating Flags Example Solution Ver 3

```
sum = 0;
for (i=0; i<10; i=i+1)
    sum = sum + A[i];
```

- sum: v1, i: v2, base address of A: v3

```
mov v1, #0
mov v2, #9 ; start with i = 9
loop: ldr al, [v3, v2, lsl #2] ; a1 = A[i]
    add v1, v1, a1 ; sum = sum + A[i]
   subs v2, v2, #1 ; decrement i
    ; update flags
    bge loop ; goto loop
```

ELEC2041 Reading Materials (Week #5)


Conditional Execution

- Recall Conditional Branch Instruction: beq, bne, bgt, bge, blt, ble, bhi, bhs, blo, bls, etc
- Almost all processors only allow branch instructions to be executed conditionally.
- However by reusing the condition evaluation hardware, ARM effectively increases number of instructions.
  - All instructions contain a condition field which determines whether the CPU will execute them.
- This removes the need for many branches, Allows very dense in-line code, without branches.
- Example:
  ```
  subs v2, v2, #1 ; Update flags
  addeq v3, v3, #2 ; add if EQ (Z = 1)
  ```
Conditional Code Example

- Convert the GCD algorithm given in this flowchart into
  1) “Normal” assembler, where only branches can be conditional.
  2) ARM assembler, where all instructions are conditional, thus improving code density.

- The only instructions you need are `cmp`, `b` and `sub`.

Conditional Code Example (Solution)

- "Normal" Assembler
  ```
  gcd    cmp r0, r1    ; reached the end?
  beq stop
  blt less        ; if r0 > r1
  sub r0, r0, r1  ; subtract r1 from r0
  bal gcd
  less  sub r1, r1, r0 ; subtract r0 from r1
  bal gcd
  stop
  ```

- ARM Conditional Assembler
  ```
  gcd    cmp   r0, r1    ; if r0 > r1
  subgt r0, r0, r1  ; subtract r1 from r0
  sublt r1, r1, r0  ; else subtract r0 ; from r1
  bne   gcd         ; reached the end?
  ```

Long Integer Addition Example

- `long int = 64 bits`

- `long int l, m, n;`

- `n = l + m;` /*Won't work in C. Still treats long int as int*/

Long Integer Addition Example (Analysis)

- We need to do the addition in two steps

```c
struct int64 {
  int lo;
  int hi;
};
l, m, n;
n.lo = m.lo + l.lo;
```

`n.hi = m.hi + l.hi + C;` How to check on carry in C?
Long Integer Carry Generation

Statement $n.lo = l.lo + m.lo$; would generate a carry $C$ if:

(Bit 31 of $l.lo = 1$) and (bit 31 of $m.lo = 1$) $\Rightarrow C = 1$

Or

(Bit 31 of $l.lo = 1$) and (bit 31 of $n.lo \neq 1$) $\Rightarrow C = 1$

Or

(Bit 31 of $m.lo = 1$) and (bit 31 of $n.lo \neq 1$) $\Rightarrow C = 1$

Long Integer Addition Example in C

```c
struct int64 { int lo; int hi; }

struct int64 ad_64(struct int64 l, struct int64 m){
    struct int64 n;
    unsigned hibitl=l.lo >> 31, hibitm=m.lo >> 31, hibitn;
    n.lo=l.lo + m.lo; hibitn=n.lo >> 31;
    n.hi=l.hi + m.hi + ((hibitl & hibitm) || (hibitl & ~hibitn) || (hibitm & ~hibitn));
    return n;
}
```

Long Integer Addition Example Compiled

```c
ad_64:
/* l.lo, l.hi, m.lo and m.hi are passed in r0-r1*/
str     lr, [sp,-#4]! ; store ret add.
mov     ip, r0, asr #31 ; hibitl (l.lo>>31)
mov    lr, r2, asr #31 ; hibitm,(m.lo>>31)
add     r0, r0, r2      ; n.lo = l.lo + m.lo
add     r1, r3, r1 ; n.hi = l.hi + m.hi
mov     r2, r0, asr #31 ; hibitn,(n.lo>>31)
tst     lr, ip ; is hibitl=hibitm ?
bne .L3
mvn     r2, r2; ~(hibitn)
tst     ip, r2 ; is hibitl=~hibitn ?
bne .L3
tst    lr, r2 ; is hibitm=~hibitn ?
beq .L2
.L3:
    mov     r2, #1 ; C = 1
.L2:
    add     r1, r1, r2 ; n.hi = l.hi + m.hi + C
    ldr     lr, [sp,#4]! ; get ret add.
    mov     pc, lr ;n.lo n.hi retuned in r0-r1
```

Long Integer Addition Example ARM

```c
ad_64:
/* l.lo, l.hi, m.lo and m.hi are passed in r0-r1*/
str     lr, [sp,-#4]! ; store ret add.
adds    r0, r0, r2 ; n.lo = l.lo + m.lo
adc     r1, r3, r1 ; n.hi = l.hi + m.hi + C
ldr     lr, [sp,#4]! ; get ret add.
mov pc, lr ;n.lo n.hi retuned in r0-r1
```

adc Is add with Carry
Long Integer Addition Example in C (GCC)

- **long long type extension in gcc**

```c
long long ad_64(long long l, long long m)
{
    long long n;
    n = m + l;
    return n;  // Long long type is 64 bits
}
```

- **ARM Compiled Code**

```
ad_64:
/* l.lo, l hi, m.lo and m.hi are passed in r0-r1*/
    adds     r0, r0, r2 ; n.lo = l.lo + m.lo
    adc      r1, r3, r1 ; n.hi = l.hi + m.hi + C
    mov     pc, lr
```

“And in Conclusion …”

- Flag Setting Instructions: `cmp, cmn, tst, teq` in ARM
- Data Processing Instructions with Flag setting Feature: `adds, subs, ands` in ARM
- Conditional Instructions: `addeq, ldreq, etc` in ARM