The ARM Assembly Language

Matteo SONZA REORDA
Dip. Automatica e Informatica
Politecnico di Torino

Introduction

- ARM is most commonly programmed using high-level languages
- Knowing the assembly language is useful
  - To understand the processor behavior
  - To optimize some critical pieces of code

Instruction categories

- ARM assembly instructions can be divided in the following categories
  - Data processing instructions
  - Data transfer instructions
  - Control flow instructions

Data processing instructions

- They perform arithmetic and logical operations on data values in registers
- They are the only instructions that modify data values

Data processing instructions

- The following rules apply:
  - All operands are 32 bits wide
  - They may be either registers or immediates
  - The result is always 32 bit wide and corresponds to a register
  - The two operands and the result are independently specified in the instruction

Classes

- Data processing instructions can be further divided in classes:
  - Arithmetic operations (ADD, ADC, SUB, SBC, RSB, RSC)
  - Bit-wise logical operations (AND, ORR, EOR, BIC)
  - Register movement operations (MOV, MVN)
  - Comparison operations (CMP, CMN, TST, TEQ)

Addition

Addition with carry

Subtraction

Classes

- Data processing instructions can be further divided in classes:
  - Arithmetic operations (ADD, ADC, SUB, SBC, RSB, RSC)
  - Bit-wise logical operations (AND, ...)

Subtraction with carry

Reverse subtraction (i.e., with operand order reversed)

Reverse subtraction with carry

... operations (M... ...)
... ...CMN, TST,

OR

EXOR

Classes

AND

... processing instructions can be ... divided in classes:
- ...metic operations (ADD, ADC, SUB, ...BC, RSB, RSC)
- Bit-wise logical operations (AND, ORR, EOR, BIC)
- Regis... ...
- Comparis... ... TEQ)

Bit clear: every '1' in the second operand clears the corresponding bit in the first

Move

Move negated: leaves in the result register the value obtained by inverting every bit in the source operand

- Data processing instructions can be further divided in classes:
  - Arithmetic operations (ADD, ADC, SUB, SBC, RSB, RSC)
  - Bit-wise logical operations (AND, ORR, EOR, BIC)
  - Register movement operations (MOV, MVN)
  - Comparison operations (CMP, CMN, TST, TEQ)

Classes

Set cc on r1 xor r2

Set cc on r2 – r1

Set cc on r1 and r2

Set cc on r1 – r2

- Data processing instructions can be further divided in classes:
  - Arithmetic operations (ADD, ADC, SUB, SBC, RSB, RSC)
  - Bit-wise logical operations (AND, ORR, EOR, BIC)
  - Register movement operations (MOV, MVN)
  - Comparison operations (CMP, CMN, TST, TEQ)

Examples

```
ADD r0, r1, r2    ; r0 := r1 + r2
ADC r0, r1, r2    ; r0 := r1 + r2 + C
AND r0, r1, r2    ; r0 := r1 and r2
MOV r0, r2        ; r0 := r2
CMP r1, r2        ; set cc on r1 - r2
ADD r3, r3, #1    ; r3 := r3 + 1
```

Immediate operands

- The second source operand may be an immediate
- The immediate
  - is represented on 8 bits
  - may be shifted left by up to 24 positions, if required

Shifted register operands

- Any operand in an instruction can be shifted before being used

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

- The whole instruction is still executed in one clock cycle

This operand can be either an immediate or a register.

Example
```
ADD r5, r5, r3, LSL r2
; r5:=r5+r3×2^{r2}
```
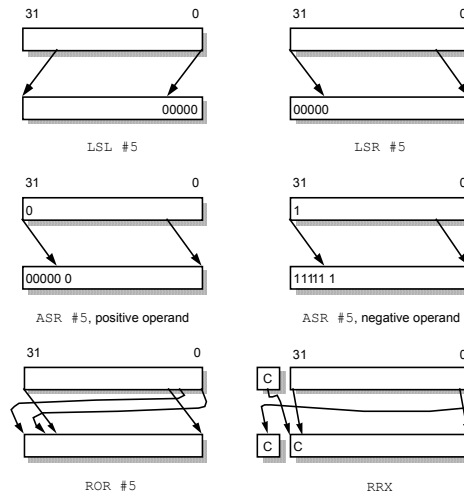
- Any operand in an instruction can be shifted

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

- The whole instruction is still executed in one clock cycle

Available shift operations

LSL #5

LSR #5

ASR #5, positive operand

ASR #5, negative operand

ROR #5

RRX

Condition codes

- Any data processing instruction can update the condition codes according to the result
- It is up to the programmer whether it has to do so, or not, by explicitly specifying his wish

Example

ADDS $\Rightarrow$ sets the condition codes

ADD $\Rightarrow$ does not set the condition codes

Multiplication

- The MUL instruction has some limitations:
  - The second operand can not be an immediate
  - The result register must not be the same as the first source register
  - Only the 32 least significant bits of the 64-bit result are stored in the result register

Data transfer instructions

- There are three classes of such instructions:
  - Single register load and store
  - Multiple register load and store
  - Single register swap

Single register load and store

- These instructions trasfer the value of a register to a memory cell (STR) or viceversa (LDR)
- The transferred value may be a 32-bit word (aligned on a 4-byte boundary), or a byte (in this case the instructions are STRB and LDRB)
- To access the memory cell, they compute an address

Register-indirect addressing

- The most common addressing mode is the register-indirect addressing

Example

```
LDR  r0, [r1]        ; r0:=mem32[r1]
STR  r0, [r1]        ; mem32[r1]:=r0
LDRB r0, [r1]        ; r0:=mem8[r1]
STRB r0, [r1]        ; mem8[r1]:=r0
```

ADR pseudo-instruction

• For the purpose of loading an address into a register, the ADR pseudo-instruction may be useful

Example

```
COPY      ADR  r1, TABLE1; r1 points to TABLE1
          ADR  r2, TABLE2; r2 points to TABLE2
...
TABLE1    ...                  ; source
TABLE2    ...                  ; destination
```

Moving vectors

```
COPY      ADR  r1, TABLE1  ; r1 points to TABLE1
          ADR  r2, TABLE2  ; r2 points to TABLE2
LOOP      LDR  r0, [r1]    ; get TABLE1 1st word
          STR  r0, [r2]    ; copy it into TABLE2
          ADD  r1, r1, #4  ; update r1
          ADD  r2, r2, #4  ; update r2
          ???              ; if more, go back to LOOP
...
TABLE1    ...                  ; source
TABLE2    ...                  ; destination
```

Base plus offset indexing

- The address may be obtained by combining
  - the value of a base register
  - an offset
- There are 3 further possibilities
  - pre-indexing
  - post-indexing
  - auto-indexing

Pre-indexing

- The address is obtained by summing an immediate to the base register
- No extra computation time is required

Example
```
LDR r0, [r1, #4] ; r0:=mem₃₂[r1+4]
```

Auto-indexing

- Allows to automatically modify the base register after using it
- No extra computation time is required

Example
```
LDR  r0, [r1, #4]!      ; r0:=mem32[r1+4]
                        ; r1:=r1+4
```

Post-indexing

- Allows to automatically modify the base register after using it without offset
- No extra computation time is required

Example
```
LDR  r0, [r1], #4       ; r0:=mem32[r1]
                        ; r1:=r1+4
```

Moving vectors (II)

```
COPY        ADR   r1, TABLE1        ; r1 points to TABLE1
            ADR   r2, TABLE2        ; r2 points to TABLE2
LOOP        LDR   r0, [r1], #4      ; get TABLE1 1st word
            STR   r0, [r2], #4      ; copy it into TABLE2
            ???                     ; if more, go back to LOOP
…
TABLE1      …                       ; source
TABLE2      …                       ; destination
```

Multiple register data transfer

- These instructions transfer any subset of registers to/from memory

Example

```
LDMIA r1,{r0, r2, r5}; r0:=mem_32[r1]
                     ; r2:=mem_32[r1+4]
                     ; r5:=mem_32[r1+8]
```

LDMIA r1,{r0, r2, r5}; $r0:=mem_{32}[r1]$
; $r2:=mem_{32}[r1+4]$
; $r5:=mem_{32}[r1+8]$

Auto-indexing

- When used with multiple register transfer instructions, the base register is updated by a value corresponding to the number of transferred bytes

Example

```
LDMIA r1!,{r0, r2, r5}    ; r0:=mem32[r1]
                          ; r2:=mem32[r1+4]
                          ; r5:=mem32[r1+8]
                          ; r1:=r1+12
```

The stack

- It is commonly adopted in microprocessor-based systems
- It may take 4 types, depending on whether it grows down or up, and whether the stack pointer points to the first empty or full cell:
  – Full ascending
  – Empty ascending
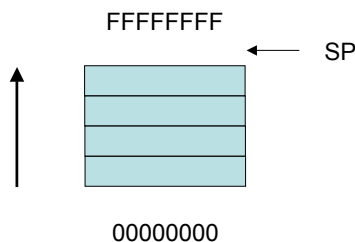 – Full descending
  – Empty descending

Full ascending

- The stack grows up through increasing memory addressing and the stack pointer points to the highest address containing a valid item
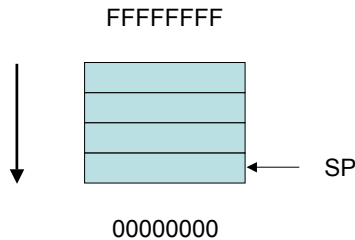
FFFFFFFF

← SP

00000000
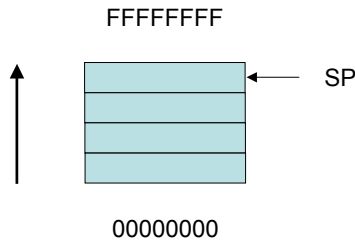
Empty ascending

- The stack grows up through increasing memory addressing and the stack pointer points to the first empty location above the stack

FFFFFFFF

← SP

00000000

Full descending

- The stack grows down through decreasing memory addressing and the stack pointer points to the lowest address containing a valid item

FFFFFFFF

SP

00000000

Empty descending

- The stack grows down through decreasing memory addressing and the stack pointer points to the first empty location below the stack

FFFFFFFF

SP

00000000
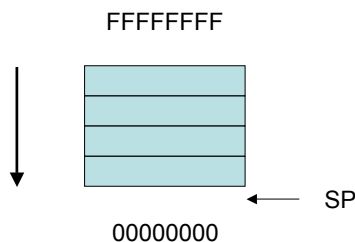
Stack implementation

- The stack can be easily implemented resorting to the auto-indexing addressing mode and the multiple register data transfer instructions with proper suffixes

Ascending		Descending	
Full	Empty	Full	Empty
STMFA LDMFA	STMEA LDMEA	STMFD LDMFD	STMED LDMED

Alternative form

- In some cases it is easier to think in terms of
  - Increment/decrement operations
  - Update operation performed before/after the move
- A set of corresponding suffixes exists
- The new set of instructions is mapped on the same instructions introduced for the stack

Stack and block copy views

		Ascending		Descending	
		Full	Empty	Full	Empty
Increment	Before	`STMIB` `STMFA`			`LDMIB` `LDMED`
	After		`STMIA` `STMEA`	`LDMIA` `LDMFD`	
Decrement	Before		`LDMDB` `LDMEA`	`STMDB` `STMFD`	
	After	`LDMDA` `LDMFA`			`STMDA` `STMED`

Example

```
LDMIA r0!,{r2-r9}
STMIA r1,{r2-r9}
```

The final result of the two instructions is to copy 8 words from the location pointed to by `r0` to that pointed to by `r1`

`r0` has been incremented by 32

Example

```
STMFD    r13!, {r2-r9}
LDMIA    r0!,{r2-r9}
STMIA    r1,{r2-r9}
LDMFD    r13!, {r2-r9}
```

The final result is the same as before, but the `r2` to `r9` registers are first saved in the stack and then restored

The stack is managed using the full descending method

Multiple register data transfers

- These instructions can be substituted by sequences of single register data transfer ones, but
  - they save code size
  - they save execution time (they are up to 4 times faster)

Control flow instructions

- They belong to several categories
  - Branches
  - Conditional branches
  - Branch and link
  - Subroutine return
  - Supervisor call

Branches

- Force the processor to execute the instruction denoted by a label

Example

```
            B       LABEL
            ...
LABEL  ...
```

Conditional branches

- Force the processor to execute a jump depending on the value of condition codes

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

Example

```
            MOV r0, r0
LOOP        …
            ADD r0, r0, #1
            CMP r0, #10
            BNE LOOP
```

Conditional execution

- Every ARM instruction may be transformed into the conditional version and its execution performed only if the condition is met

Example
```
CMP    r0, #5        ; if (r0!= 5){
ADDNE  r1, r1, r0    ;    r1:=r1+r0-r2
SUBNE  r1, r1, r2    ; }
```

Example

- Conditional execution may produce very compact code

Example
```
; if ((a==b)&&(c==d)) e++;

  CMP    r0, r1
  CMPEQ  r2, r3
  ADDEQ  r4, r4, #1
```

Speed

- The use of conditional instructions is generally convenient when the conditional clause is composed of 3 instructions or less

Branch and link

- This instruction allows implementing the call to a procedure
- It performs as a branch, but also saves the address of the following instruction in `r14`

Nested procedures

- If a procedure calls another, it must first save r14, normally into the stack
- If other registers must be saved, a multiple register store instruction may be used

Example
```
      BL    SUB1
      …
SUB1  STMFD      r13!, {r0-r2,r14}
      BL         SUB2
      …
SUB2  …
```

Return from subroutine

- It requires restoring the pc with the saved value
- This can be done
  - With a MOV instruction, if the return address is in r14
    ```
     MOV  pc, r14
    ```
  - With a multiple register load instruction, if the return address is in the stack, and other registers should also be restored

Example

```
SUB1 STMFD    r13!, {r0-r2,r14}
     BL       SUB2
     …
     LDMFD    r13!, {r0-r2,pc}
```

Supervisor calls

- They allow calling system procedures (e.g., to perform I/O operations) that are executed in protected mode
- They are based on the `SWI` (SoftWare Interrupt) instruction

Example
```
SWI  SWI_WriteC; output one char
```

Jump tables

- Can be easily and efficiently implemented in the ARM assembly language

Example
```
            BL     JUMPTAB
            …
JUMPTAB     ADR    r1, SUBTAB
            CMP    r0, #SUBMAX
            LDRLS  PC, [r1, r0, LSL #2]
            B      ERROR
SUBTAB DCD  SUB0
            DCD    SUB1
            DCD    SUB2
            …
```

A complete program

```
            AREA          HelloW, CODE, READONLY
SWI_WriteC                EQU          &0
SWI_Exit                  EQU          &11
            ENTRY
START       ADR           r1, TEXT
LOOP        LDRB          r0, [r1], #1
            CMP           r0, #0
            SWINE         SWI_WriteC
            BNE           LOOP
            SWI           SWI_Exit
TEXT        =             "Hello World", &0a, &0d, 0
            END
```