ARM Assembly Language Programming

- Outline:
  - the ARM instruction set
  - writing simple programs
  - examples

The ARM instruction set

- ARM instructions fall into three categories:
  - data processing instructions
    - operate on values in registers
  - data transfer instructions
    - move values between memory and registers
  - control flow instructions
    - change the program counter (PC)

Data processing instructions

- All operands are 32-bits wide and either:
  - come from registers, or
  - are literals (‘immediate’ values) specified in the instruction
- The result, if any, is 32-bits wide and goes into a register
  - except long multiplies generate 64-bit results
- All operand and result registers are independently specified

Data processing instructions

- Examples:
  
  ```assembly
  ADD   r0, r1, r2 ; r0 := r1 + r2
  SUB   r0, r1, #2  ; r0 := r1 - 2
  
  Note:
  - everything after the ‘;’ is a comment
    - it is there solely for the programmer’s convenience
  - the result register (r0) is listed first
  ```

Arithmetic operations:

```assembly
ADD   r0, r1, r2 ; r0 := r1 + r2
ADC   r0, r1, r2 ; r0 := r1 + r2 + C
SUB   r0, r1, r2 ; r0 := r1 - r2
SBC   r0, r1, r2 ; r0 := r1 - r2 + C - 1
RSB   r0, r1, r2 ; r0 := r2 - r1
RSC   r0, r1, r2 ; r0 := r2 - r1 + C - 1
  
  • C is the C bit in the CPSR
  • the operation may be viewed as unsigned or 2’s complement signed
```
Data processing instructions

- **Bit-wise logical operations:**
  
  ```
  AND r0, r1, r2 ; r0 := r1 and r2
  ORR r0, r1, r2 ; r0 := r1 or r2
  EOR r0, r1, r2 ; r0 := r1 xor r2
  BIC r0, r1, r2 ; r0 := r1 and not r2
  ```

  - the specified Boolean logic operation is performed on each bit from 0 to 31
  - **BIC** stands for ‘bit clear’
  - each ‘1’ in r2 clears the corresponding bit in r1

- **Register movement operations:**
  
  ```
  MOV r0, r2 ; r0 := r2
  MVN r0, r2 ; r0 := not r2
  ```

  - **MVN** stands for ‘move negated’
  - there is no first operand (r1) specified as these are unary operations

- **Comparison operations:**
  
  ```
  CMP r1, r2 ; set cc on r1 - r2
  CMN r1, r2 ; set cc on r1 + r2
  TST r1, r2 ; set cc on r1 and r2
  TEQ r1, r2 ; set cc on r1 xor r2
  ```

  - these instructions just affect the condition codes (N, Z, C, V) in the CPSR
  - there is no result register (r0)

- **Immediate operands**
  
  ```
  ADD r3, r3, #1 ; r3 := r3 + 1
  AND r8, r7, #ff ; r8 := r7[7:0]
  ```

  - # indicates an immediate value
  - & indicates hexadecimal notation
  - C-style notation (#0xff) is also supported
  - allowed immediate values are (in general): (0 \rightarrow 255) \times 2^n

- **Shifted register operands**
  
  ```
  ADD r3, r2, r1, LSL #3 ; r3 := r2+r1<<3
  ```

  - by a constant number of bit positions
  - or by a register-specified number of bits:
    ```
    ADD r5, r5, r3, LSL r2 ; r5 := r3<<r2
    ```

  - **LSL, LSR** mean ‘logical shift left, right’
  - **ASL, ASR** mean ‘arithmetic shift left, right’
  - **ROR** means ‘rotate right’
  - **RRX** means ‘rotate right extended’ by 1 bit

Data processing instructions

- **Register movement operations:**
  
  ```
  MOV r0, r2 ; r0 := r2
  MVN r0, r2 ; r0 := not r2
  ```

  - **MVN** stands for ‘move negated’
  - there is no first operand (r1) specified as these are unary operations

- **Comparison operations:**
  
  ```
  CMP r1, r2 ; set cc on r1 - r2
  CMN r1, r2 ; set cc on r1 + r2
  TST r1, r2 ; set cc on r1 and r2
  TEQ r1, r2 ; set cc on r1 xor r2
  ```

  - these instructions just affect the condition codes (N, Z, C, V) in the CPSR
  - there is no result register (r0)

- **Immediate operands**
  
  ```
  ADD r3, r3, #1 ; r3 := r3 + 1
  AND r8, r7, #ff ; r8 := r7[7:0]
  ```

  - # indicates an immediate value
  - & indicates hexadecimal notation
  - C-style notation (#0xff) is also supported
  - allowed immediate values are (in general): (0 \rightarrow 255) \times 2^n

- **Shifted register operands**
  
  ```
  ADD r3, r2, r1, LSL #3 ; r3 := r2+r1<<3
  ```

  - by a constant number of bit positions
  - or by a register-specified number of bits:
    ```
    ADD r5, r5, r3, LSL r2 ; r5 := r3<<r2
    ```

  - **LSL, LSR** mean ‘logical shift left, right’
  - **ASL, ASR** mean ‘arithmetic shift left, right’
  - **ROR** means ‘rotate right’
  - **RRX** means ‘rotate right extended’ by 1 bit

**ARM shift operations**

- **LSL #5**
  - Left Shift Logical
- **LSR #5**
  - Left Shift Arithmetic
- **ASL #5**
  - Arithmetic Shift Left
- **ASR #5**
  - Arithmetic Shift Right
- **ROR #5**
  - Rotate Right
- **RRX**
  - Rotate Right Extended by 1 bit
Data processing instructions

- Setting the condition codes
  - all data processing instructions may set the condition codes.
  - the comparison operations always do so

For example, here is code for a 64-bit add:

```
ADDS r2, r2, r0 ; 32-bit carry-out -> C
ADC r3, r3, r1 ; added into top 32 bits
S means ‘Set condition codes’
• the primary use of the condition codes is in control flow - see later
```

Data processing instructions

- Multiplication
  - ARM has special multiply instructions
    MUL r4, r3, r2 ; r4 := (r3 x r2)[31:0]
    - only the bottom 32 bits are returned
    - immediate operands are not supported
    - multiplication by a constant is usually best done with a short series of adds and subtracts with shifts
  - there is also a ‘multiply-accumulate’ form:
    MLA r4, r3, r2, r1 ; r4 := (r3xr2+r1)[31:0]
  - 64-bit result forms are supported too

The ARM instruction set

- ARM instructions fall into three categories:
  - data processing instructions
    - operate on values in registers
  → data transfer instructions
  - move values between memory and registers
  - control flow instructions
    - change the program counter (PC)

Data transfer instructions

- The ARM has 3 types of data transfer instruction:
  - single register loads and stores
    - flexible byte, half-word and word transfers
  - multiple register loads and stores
    - less flexible, multiple words, higher transfer rate
  - single register - memory swap
    - mainly for system use, so ignore for now

Data transfer instructions

- Initializing an address pointer
  - any register can be used for an address
  - the assembler has special ‘pseudo instructions’ to initialise address registers:
    ADR r1, TABLE1 ; r1 points to TABLE1
    ..
    TABLE1 ; LABEL
    - ADR will result in a single ARM instruction
    ADDR r1, TABLE1
    - ADDR will handle cases that ADR can’t

Data transfer instructions

- Addressing memory
  - all ARM data transfer instructions use register indirect addressing.
Examples of load and store instructions:

```
LDR r0, [r1] ; r0 := mem[r1]
STR r0, [r1] ; mem[r1] := r0
```

- therefore before any data transfer is possible:
  - a register must be initialized with an address close to the target
Data transfer instructions

- **Single register loads and stores**
  - The simplest form is just register indirect:
    \[ \text{LDR } r0, [r1] \quad ; \quad r0 := \text{mem}[r1] \]
  - This is a special form of ‘base plus offset’:
    \[ \text{LDR } r0, [r1,#4] \quad ; \quad r0 := \text{mem}[r1+4] \]
  - The offset is within ±4 Kbytes
  - Auto-indexing is also possible:
    \[ \text{LDR } r0, [r1,#4]! ; r0 := \text{mem}[r1+4] ; r1 := r1 + 4 \]

- **Another form uses post-indexing**
  \[ \text{LDR } r0, [r1],#4 ; r0 := \text{mem}[r1] ; r1 := r1 + 4 \]

- Finally, a byte or half-word can be loaded instead of a word (with some restrictions):
  \[ \text{LDRB } r0, [r1] \quad ; r0 := \text{mem}_8[r1] \]
  \[ \text{LDRSH } r0, [r1] \quad ; r0 := \text{mem}_{16}[r1](\text{signed}) \]

- Stores (STR) have the same forms

Multiple transfer instructions

- **Multiple register loads and stores**
  - ARM also supports instructions which transfer several registers:
    \[ \text{LDMIA } r1, \{r0,r2,r5\} \quad ; \quad r0 := \text{mem}[r1] \]
    \[ ; r2 := \text{mem}[r1+4] \]
    \[ ; r5 := \text{mem}[r1+8] \]
  - The \{..\} list may contain any or all of \text{r0 - r15}
  - The lowest register always uses the lowest address, and so on, in increasing order
    - It doesn’t matter how the registers are ordered in \{..\}

Multiple register transfer addressing modes

Stack and block copy views of the load and store multiple instructions

- **Multiple register loads and stores (.ctd)**
  - Another form uses post-indexing
    \[ \text{LDR } r0, [r1],#4 ; r0 := \text{mem}[r1] \]
    \[ ; r1 := r1 + 4 \]
  - Finally, a byte or half-word can be loaded instead of a word (with some restrictions):
    \[ \text{LDRB } r0, [r1] ; r0 := \text{mem}_8[r1] \]
    \[ \text{LDRSH } r0, [r1] ; r0 := \text{mem}_{16}[r1](\text{signed}) \]
  - Stores (STR) have the same forms
The ARM instruction set

- ARM instructions fall into three categories:
  - data processing instructions
    - operate on values in registers
  - data transfer instructions
    - move values between memory and registers
  - control flow instructions
    - change the program counter (PC)

Control flow instructions

- Control flow instructions just switch execution around the program:
  - B LABEL
    - .. ; these instructions are skipped
  - LABEL ..
    - normal execution is sequential
    - branches are used to change this
      - to move forwards or backwards
    - Note: data ops and loads can also change the PC!

Conditional branches

- sometimes whether or not a branch is taken depends on the condition codes:
  - MOV r0, #0 ; initialise counter
  - LOOP ..
    - ADD r0, r0, #1 ; increment counter
  - CMP r0, #10 ; compare with limit
  - BNE LOOP ; repeat if not equal
    - here the branch depends on how CMP sets Z

Branch conditions

- Branch Interpretation Normal use
  - B  Unconditional Always takes this branch
  - BL  Always Always takes 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
  - BHS  Higher or same Unsigned comparison gave higher or same
  - BVS  Overflow set Signed integer operation; overflow occurred
  - BVC  Overflow clear Signed integer operation; no overflow occurred
  - BNEQ  Not equal Signed integer comparison gave different
  - BLE  Less or equal Signed integer comparison gave less than or equal
  - BGT  Greater than Signed integer comparison gave greater than
  - BLT  Less than Signed integer comparison gave less than
  - BHI  Higher Unsigned comparison gave higher
  - BLS  Lower or same Unsigned comparison gave lower or same

Conditional execution

- an unusual ARM feature is that all instructions may be conditional:
  - CMP r0, #5 ; if (r0 != 5) {
    - ADDNE r1, r1, r0 ; r1 := r1 + r0 - r2
    - SUBNE r1, r1, r2 ;
  }
  - this removes the need for some short branches
    - improving performance and code density

Branch and link

- ARM’s subroutine call mechanism
- saves the return address in r14
  - BL SUBR ; branch to SUBR
    - .. ; return to here
  - SUBR .. ; subroutine entry point
    - MOV pc, r14 ; return
      - note the use of a data processing instruction for return
    - r14 is often called the link register (lr)
      - the only special register use other than the PC

Conditional execution

- an unusual ARM feature is that all instructions may be conditional:
  - CMP r0, #5 ; if (r0 != 5) {
    - ADDONE r1, r1, r0 ; r1 := r1 + r0 - r2
    - SUBNE r1, r1, r2 ;
  }
  - this removes the need for some short branches
    - improving performance and code density
Control flow instructions

- **Supervisor calls**
  - these are calls to operating system functions such as input and output:

    ```assembly
    SWI SWI_Angel ; call Angel monitor
    ```
  - the range of available calls is system dependent
    - Angel uses a single SWI number with the system function specified in r0

Writing simple programs

- **Assembler details to note:**
  - AREA - declaration of code area
  - EQU - initialising constants
    - used here to define SWI number
  - ENTRY - code entry point
  - =, DCB, DCD - ways to initialise memory
  - END - the end of the program source
  - labels are aligned left
    - opcodes are indented

Examples

- **‘Hello World’ assembly program:**

  ```assembly
  AREA     HelloW,CODE,READONLY ; declare area
  SWI_Angel EQU 0x123456 ; Angel SWI number
  ENTRY
  START   ADR     r1, TEXT-1 ; r1 -> “Hello World” -1
  LOOP    MOV     r0, #0x3 ; Angel write char in [r1]
           LDRB    r2, [r1,#1]! ; get the next byte
           CMP     r2, #0 ; check for text end
           SWINE   SWI_Angel    ; if not end print ..
           BNE     LOOP         ;  .. and loop back
  MOV     r0, #0x18 ; Angel exception call
  LDR     r1, =0x20026 ;    Exit reason
  MOV     r0, #0x3 ; Angel write char in [r1]
  SWI     SWI_Angel    ; end of execution
  TEXT    = "Hello World",0xa,0xd,0
  END
  ```

Examples

- **Subroutine to print r2 in hexadecimal**

  ```assembly
  HexOut MOV   r3, #8 ; nibble count = 8
           ADR   r1, ChOut       ; buffer for write char
  LOOP   MOV   r0, r2, LSR #28 ; get top nibble
           CMP   r0, #9 ; 0-9 or A-F?
           ADDGT r0, r0, #"A"-10 ; ASCII alphabetic
           ADDLE r0, r0, #"0" ; ASCII numeric
           STR   r0, [r1]        ; store ASCII in buffer
           MOV   r0, #0x3 ; Angel write char in [r1]
           SWI   SWI_Angel    ; print character
           MOV   r2, r2, LSL #4 ; shift left one nibble
           SUBS  r3, r3, #1 ; decrement nibble count
           BNE   LOOP            ; if more do next nibble
           MOV   pc, r14 ; ... else return
  ChOut   DCD   0
  ```