ARM Assembler

Data Movement

Memory Access

- Load Register from memory

  LDR$\langle cc \rangle$ Rd, $\langle op2 \rangle$

  $\langle cc \rangle$: MAR $\leftarrow \langle op2 \rangle$

  $\langle cc \rangle$: MBR $\leftarrow$ M(MAR)

  $\langle cc \rangle$: Rd $\leftarrow$ MBR

- Store Register in memory

  STR$\langle cc \rangle$ Rs, $\langle op2 \rangle$

  $\langle cc \rangle$: MAR $\leftarrow \langle op2 \rangle$

  $\langle cc \rangle$: MBR $\leftarrow$ Rs

  $\langle cc \rangle$: M(MAR) $\leftarrow$ MBR

- Memory Reference must be 32-bit word aligned otherwise a Data Abort Exception will occur use the ALIGN directive to force alignment

Load / Store Byte

- Load Register with unsigned Byte from memory

  LDR⟨*cc*⟩B Rd, ⟨*op2*⟩

  ⟨*cc*⟩: MAR ← ⟨*op2*⟩
  ⟨*cc*⟩: MBR ← M(MAR)
  ⟨*cc*⟩: Rd(7:0) ← MBR
  ⟨*cc*⟩: Rd(31:8) ← 0

- Load Register with Signed Byte from memory

  LDR⟨*cc*⟩SB Rd, ⟨*op2*⟩

  ⟨*cc*⟩: MAR ← ⟨*op2*⟩
  ⟨*cc*⟩: MBR ← M(MAR)
  ⟨*cc*⟩: Rd(7:0) ← MBR
  ⟨*cc*⟩: Rd(31:8) ← Rd(7)

- Store Register in a Byte of memory

  STR⟨*cc*⟩B Rs, ⟨*op2*⟩

  ⟨*cc*⟩: MAR ← ⟨*op2*⟩
  ⟨*cc*⟩: MBR ← Rs
  ⟨*cc*⟩: M(MAR) ← Rs(7:0)

Load / Store Halfword

- **Does not work in the ARMulator**
- An ARM word is 32-bits, so a Halfword is 16-bits
- Memory Reference must be Halfword aligned
- Load Register with unsigned Halfword from memory

  LDR⟨*cc*⟩H Rd, ⟨*op2*⟩

  ⟨*cc*⟩: MAR ← ⟨*op2*⟩
  ⟨*cc*⟩: MBR ← M(MAR)
  ⟨*cc*⟩: Rd(15:0) ← MBR
  ⟨*cc*⟩: Rd(31:16) ← 0
- Load Register with Signed Halfword from memory

  LDR⟨*cc*⟩SH Rd, ⟨*op2*⟩

  ⟨*cc*⟩: MAR ← ⟨*op2*⟩
  ⟨*cc*⟩: MBR ← M(MAR)
  ⟨*cc*⟩: Rd(15:0) ← MBR
  ⟨*cc*⟩: Rd(31:16) ← Rd(15)
- Store Register in a Halfword of memory

  STR⟨*cc*⟩H Rs, ⟨*op2*⟩

  ⟨*cc*⟩: MAR ← ⟨*op2*⟩
  ⟨*cc*⟩: MBR ← Rs
  ⟨*cc*⟩: M(MAR) ← MBR(15:0)

Program: move16.s

```
1.  ; 16bit data transfer
2.
3.          TTL     move16 – 16-bit data transfer
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDRB    R1, Value       ; Load value
9.          STR     R1, Result      ; Sore it again
10.         SWI     &11             ; exit()
11.
12. Value   DCW     &C123           ; Source value to be moved
13.         ALIGN                   ; Alling next word
14. Result  DCW     0               ; Reserve space for result
15.
16.         END
```

Program: move16.s

```
1.  ; 16bit data transfer
2.
3.          TTL     move16 – 16-bit data transfer
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDRB    R1, Value       ; Load value
9.          STR     R1, Result      ; Sore it again
10.         SWI     &11             ; exit()
11.
12. Value   DCW     &C123           ; Source value to be moved
13.         ALIGN                   ; Alling next word
14. Result  DCW     0               ; Reserve space for result
15.
16.         END
```

TTL Define Program *Title*

Program: move16.s

```
1.  ; 16bit data transfer
2.
3.          TTL     move16 – 16-bit data transfer
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDRB    R1, Value       ; Load value
9.          STR     R1, Result      ; Sore it again
10.         SWI     &11             ; exit()
11.
12. Value   DCW     &C123           ; Source value to be moved
13.         ALIGN                   ; Alling next word
14. Result  DCW     0               ; Reserve space for result
15.
16.         END
```

AREA Label Program Area

Code or Data space; Read Only or Read / Write

Program: move16.s

```
1.  ; 16bit data transfer
2.
3.          TTL     move16 – 16-bit data transfer
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDRB    R1, Value       ; Load value
9.          STR     R1, Result      ; Sore it again
10.         SWI     &11             ; exit()
11.
12. Value   DCW     &C123           ; Source value to be moved
13.         ALIGN                   ; Alling next word
14. Result  DCW     0               ; Reserve space for result
15.
16.         END

ENTRY       Define Program Entry Point
```

Program: move16.s

```
1.  ; 16bit data transfer
2.
3.          TTL     move16 – 16-bit data transfer
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDRB    R1, Value       ; Load value
9.          STR     R1, Result      ; Sore it again
10.         SWI     &11             ; exit()
11.
12. Value   DCW     &C123           ; Source value to be moved
13.         ALIGN                   ; Alling next word
14. Result  DCW     0               ; Reserve space for result
15.
16.         END
```

Main Label the memory address

Debug will place breakpoint at *Main*

Program: move16.s

```
1.  ; 16bit data transfer
2.
3.          TTL     move16 – 16-bit data transfer
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDRB    R1, Value       ; Load value
9.          STR     R1, Result      ; Sore it again
10.         SWI     &11             ; exit()
11.
12. Value   DCW     &C123           ; Source value to be moved
13.         ALIGN                   ; Alling next word
14. Result  DCW     0               ; Reserve space for result
15.
16.         END
```

SWI Software Interrupt — Call the Operating System

`exit()`

Program: move16.s

```
1.  ; 16bit data transfer
2.
3.          TTL     move16 – 16-bit data transfer
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDRB    R1, Value       ; Load value
9.          STR     R1, Result      ; Sore it again
10.         SWI     &11             ; exit()
11.
12. Value   DCW     &C123           ; Source value to be moved
13.         ALIGN                   ; Alling next word
14. Result  DCW     0               ; Reserve space for result
15.
16.         END
```

& Define a Hexadecimal value

Program: move16.s

```
1.  ; 16bit data transfer
2.
3.          TTL     move16 – 16-bit data transfer
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDRB    R1, Value       ; Load value
9.          STR     R1, Result      ; Sore it again
10.         SWI     &11             ; exit()
11.
12. Value   DCW     &C123           ; Source value to be moved
13.         ALIGN                   ; Alling next word
14. Result  DCW     0               ; Reserve space for result
15.
16.         END
```

DCW Define a 16-bit data value

Program: move16.s

```
1.  ; 16bit data transfer
2.
3.          TTL     move16 – 16-bit data transfer
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDRB    R1, Value       ; Load value
9.          STR     R1, Result      ; Sore it again
10.         SWI     &11             ; exit()
11.
12. Value   DCW     &C123           ; Source value to be moved
13.         ALIGN                   ; Alling next word
14. Result  DCW     0               ; Reserve space for result
15.
16.         END
```

ALIGN Align data item on 32-bit word boundary

Program: move16.s

```
1.  ; 16bit data transfer
2.
3.          TTL     move16 – 16-bit data transfer
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDRB    R1, Value       ; Load value
9.          STR     R1, Result      ; Sore it again
10.         SWI     &11             ; exit()
11.
12. Value   DCW     &C123           ; Source value to be moved
13.         ALIGN                   ; Alling next word
14. Result  DCW     0               ; Reserve space for result
15.
16.         END

END         End of program source
```

Program: move16.s

```
1.  ; 16bit data transfer
2.
3.          TTL     move16 – 16-bit data transfer
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDRB    R1, Value       ; Load value
9.          STR     R1, Result      ; Sore it again
10.         SWI     &11             ; exit()
11.
12. Value   DCW     &C123           ; Source value to be moved
13.         ALIGN                   ; Alling next word
14. Result  DCW     0               ; Reserve space for result
15.
16.         END
```

Bug Assembler can only find *syntax* errors

You have to find the *logical* errors

Data Movement

- `MOV` Move Data
  `MOVS` Move Data and Set Zero and Negative flags
  `MOV`$\langle cc \rangle$ Move Data if $\langle cc \rangle$
- `MOV`$\langle cc \rangle\langle S \rangle$ `R`d`,` $\langle op1 \rangle$

  $\langle cc \rangle$: ALU $\leftarrow \langle op1 \rangle$

  $\langle cc \rangle$: Rd $\leftarrow$ ALU

  $\langle S \rangle\langle cc \rangle$: CPSR $\leftarrow$ ALU(Flags)
- Move and Negate Data

  `MVN`$\langle cc \rangle\langle S \rangle$ `R`d`,` $\langle op1 \rangle$

  $\langle cc \rangle$: ALU $\leftarrow \overline{\langle op1 \rangle}$

  $\langle cc \rangle$: Rd $\leftarrow$ ALU

  $\langle S \rangle\langle cc \rangle$: CPSR $\leftarrow$ ALU(Flags)
- Rd is the destination (must be a register)
- $\langle op1 \rangle$ is the source

Program: invert.s

```
1.  ; Find the one's compliment (inverse) of a number
2.
3.          TTL     invert.s – one's complement
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDR     R1, Value       ; Load number to be processed
9.          MVN     R1, R1          ; Invert (not) the value
10.         STR     R1, Result      ; Store the result
11.         SWI     &11             ; exit()
12.
13. Value   DCD     &C123           ; Value to be complemented
14. Result  DCD     0               ; Reserve space for result
15.
16.         END
```

Program: invert.s

```
1.  ; Find the one's compliment (inverse) of a number
2.
3.          TTL     invert.s – one's complement
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDR     R1, Value       ; Load number to be processed
9.          MVN     R1, R1          ; Invert (not) the value
10.         STR     R1, Result      ; Store the result
11.         SWI     &11             ; exit()
12.
13. Value   DCD     &C123           ; Value to be complemented
14. Result  DCD     0               ; Reserve space for result
15.
16.         END
```

Labels Used to access memory directly

Program: invert.s

```
1.  ; Find the one's compliment (inverse) of a number
2.
3.          TTL     invert.s – one's complement
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDR     R1, Value       ; Load number to be processed
9.          MVN     R1, R1          ; Invert (not) the value
10.         STR     R1, Result      ; Store the result
11.         SWI     &11             ; exit()
12.
13. Value   DCD     &C123           ; Value to be complemented
14. Result  DCD     0               ; Reserve space for result
15.
16.         END
```

DCD Used to define (and initialise) memory values

No need for `ALIGN` as `DCD` defines 32-bit values

Program: invert.s

```
1.  ; Find the one's compliment (inverse) of a number
2.
3.          TTL     invert.s – one's complement
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDR     R1, Value   ; Load number to be processed
9.          MVN     R1, R1      ; Invert (not) the value
10.         STR     R1, Result  ; Store the result
11.         SWI     &11         ; exit()
12.
13. Value   DCD     &C123       ; Value to be complemented
14. Result  DCD     0           ; Reserve space for result
15.
16.         END
```

MNV Move and Negate

Uses same register for *Source*$_1$ and *Destination*

Arithmetic

- Addition

  ADD$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

  $\langle cc \rangle$: ALU $\leftarrow$ R$n + \langle op1 \rangle$
  $\langle cc \rangle$: Rd $\leftarrow$ ALU
  $\langle S \rangle\langle cc \rangle$: CPSR $\leftarrow$ ALU(Flags)

- Subtraction

  SUB$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

  $\langle cc \rangle$: ALU $\leftarrow$ R$n - \langle op1 \rangle$
  $\langle cc \rangle$: Rd $\leftarrow$ ALU
  $\langle S \rangle\langle cc \rangle$: CPSR $\leftarrow$ ALU(Flags)

- Multiplication

  MUL$\langle cc \rangle\langle S \rangle$ Rd, Rn, Rs

  $\langle cc \rangle$: ALU $\leftarrow$ R$n \times$ Rs
  $\langle cc \rangle$: Rd $\leftarrow$ ALU
  $\langle S \rangle\langle cc \rangle$: CPSR $\leftarrow$ ALU(Flags)

  Multiply two 16-bit values (Rn and Rs) producing a 32-bit result (Rd)

- Division

  There is **no** division instruction

Program: add2.s

```
1.  ; Add two numbers and store the result
    ...
7.  Main
8.          LDR     R0, =Value1     ; R0 = &Value1
9.          LDR     R1, [R0]        ; R1 = *R0
10.         ADD     R0, R0, #0x4    ; R0++
11.         LDR     R2, [R0]        ; R2 = *R0
12.         ADD     R1, R1, R2      ; R1 = R1 + R2
13.         LDR     R0, =Result     ; R0 = &Result
14.         STR     R1, [R0]        ; *R0 = R1
15.         SWI     &11             ; exit(0)
16.
17. Value1  DCD     &37E3C123
18. Value2  DCD     &367402AA
19. Result  DCD     0
    ...
```

Program: add2.s

```
1.  ; Add two numbers and store the result
    ...
7.  Main
8.          LDR     R0, =Value1     ; R0 = &Value1
9.          LDR     R1, [R0]        ; R1 = *R0
10.         ADD     R0, R0, #0x4    ; R0++
11.         LDR     R2, [R0]        ; R2 = *R0
12.         ADD     R1, R1, R2      ; R1 = R1 + R2
13.         LDR     R0, =Result     ; R0 = &Result
14.         STR     R1, [R0]        ; *R0 = R1
15.         SWI     &11             ; exit(0)
16.
17. Value1  DCD     &37E3C123
18. Value2  DCD     &367402AA
19. Result  DCD     0
    ...
```

... Lines of no interest are ignored

Program: add2.s

```
1.  ; Add two numbers and store the result
    ...
7.  Main
8.          LDR     R0, =Value1     ; R0 = &Value1
9.          LDR     R1, [R0]        ; R1 = *R0
10.         ADD     R0, R0, #0x4    ; R0++
11.         LDR     R2, [R0]        ; R2 = *R0
12.         ADD     R1, R1, R2      ; R1 = R1 + R2
13.         LDR     R0, =Result     ; R0 = &Result
14.         STR     R1, [R0]        ; *R0 = R1
15.         SWI     &11             ; exit(0)
16.
17. Value1  DCD     &37E3C123
18. Value2  DCD     &367402AA
19. Result  DCD     0
    ...
```

ADD Same register for $Source_1$ and *Destination*

Program: add2.s

```
1.  ; Add two numbers and store the result
    ...
7.  Main
8.          LDR     R0, =Value1     ; R0 = &Value1
9.          LDR     R1, [R0]        ; R1 = *R0
10.         ADD     R0, R0, #0x4    ; R0++
11.         LDR     R2, [R0]        ; R2 = *R0
12.         ADD     R1, R1, R2      ; R1 = R1 + R2
13.         LDR     R0, =Result     ; R0 = &Result
14.         STR     R1, [R0]        ; *R0 = R1
15.         SWI     &11             ; exit(0)
16.
17. Value1  DCD     &37E3C123
18. Value2  DCD     &367402AA
19. Result  DCD     0
    ...
```

=label Load address of *label* into R0

Program: add2.s

```
1.  ; Add two numbers and store the result
    ...
7.  Main
8.          LDR     R0, =Value1     ; R0 = &Value1
9.          LDR     R1, [R0]        ; R1 = *R0
10.         ADD     R0, R0, #0x4    ; R0++
11.         LDR     R2, [R0]        ; R2 = *R0
12.         ADD     R1, R1, R2      ; R1 = R1 + R2
13.         LDR     R0, =Result     ; R0 = &Result
14.         STR     R1, [R0]        ; *R0 = R1
15.         SWI     &11             ; exit(0)
16.
17. Value1  DCD     &37E3C123
18. Value2  DCD     &367402AA
19. Result  DCD     0
    ...
```

LDR Load data from memory pointed to by R0

Program: add2.s

```
1.  ; Add two numbers and store the result
    ...
7.  Main
8.          LDR     R0, =Value1     ; R0 = &Value1
9.          LDR     R1, [R0]        ; R1 = *R0
10.         ADD     R0, R0, #0x4    ; R0++
11.         LDR     R2, [R0]        ; R2 = *R0
12.         ADD     R1, R1, R2      ; R1 = R1 + R2
13.         LDR     R0, =Result     ; R0 = &Result
14.         STR     R1, [R0]        ; *R0 = R1
15.         SWI     &11             ; exit(0)
16.
17. Value1  DCD     &37E3C123
18. Value2  DCD     &367402AA
19. Result  DCD     0
    ...
```

ADD Increment pointer in R0 by a word (4 bytes)

Program: add2.s

```
1.    ; Add two numbers and store the result
      ...
7.    Main
8.          LDR    R0, =Value1     ; R0 = &Value1
9.          LDR    R1, [R0]        ; R1 = *R0
10.         ADD    R0, R0, #0x4    ; R0++
11.         LDR    R2, [R0]        ; R2 = *R0
12.         ADD    R1, R1, R2      ; R1 = R1 + R2
13.         LDR    R0, =Result     ; R0 = &Result
14.         STR    R1, [R0]        ; *R0 = R1
15.         SWI    &11             ; exit(0)
16.
17.   Value1 DCD    &37E3C123
18.   Value2 DCD    &367402AA
19.   Result DCD    0
      ...
```

ADD/LDR — No need for ADD instructions if LDR uses post-index addressing: [R0], #0x4 or *(R0++) in C

Program: add2.s

```
1.  ; Add two numbers and store the result
    ...
7.  Main
8.          LDR     R0, =Value1     ; R0 = &Value1
9.          LDR     R1, [R0]        ; R1 = *R0
10.         ADD     R0, R0, #0x4    ; R0++
11.         LDR     R2, [R0]        ; R2 = *R0
12.         ADD     R1, R1, R2      ; R1 = R1 + R2
13.         LDR     R0, =Result     ; R0 = &Result
14.         STR     R1, [R0]        ; *R0 = R1
15.         SWI     &11             ; exit(0)
16.
17. Value1  DCD     &37E3C123
18. Value2  DCD     &367402AA
19. Result  DCD     0
    ...
```

STR Store data indirect (at memory pointed to by R0)

Program: add2.s

```
1.  ; Add two numbers and store the result
    . . .
7.  Main
8.          LDR     R0, =Value1     ; R0 = &Value1
9.          LDR     R1, [R0]        ; R1 = *R0
10.         ADD     R0, R0, #0x4    ; R0++
11.         LDR     R2, [R0]        ; R2 = *R0
12.         ADD     R1, R1, R2      ; R1 = R1 + R2
13.         LDR     R0, =Result     ; R0 = &Result
14.         STR     R1, [R0]        ; *R0 = R1
15.         SWI     &11             ; exit(0)
16.
17. Value1  DCD     &37E3C123
18. Value2  DCD     &367402AA
19. Result  DCD     0
    . . .
```

Comments These are *bad* comments

Comments should say *why* not *what*

Program: shiftleft.s

```
1.  ; Shift Left one bit
2.
3.          TTL     shiftleft.s
4.          AREA    Program, CODE, READONLY
5.          ENTRY
6.
7.  Main
8.          LDR     R1, Value           ; Load the value to be shifted
9.          MOV     R1, R1, LSL #0x1    ; Shift Left one bit
10.         STR     R1, Result          ; Store the result
11.         SWI     &11                 ; exit
12.
13. Value   DCD     &4242               ; Value to be shifted
14. Result  DCD     0                   ; Space to store result
15.
16.         END
```

LSL Logical Shift Left by 1 bit