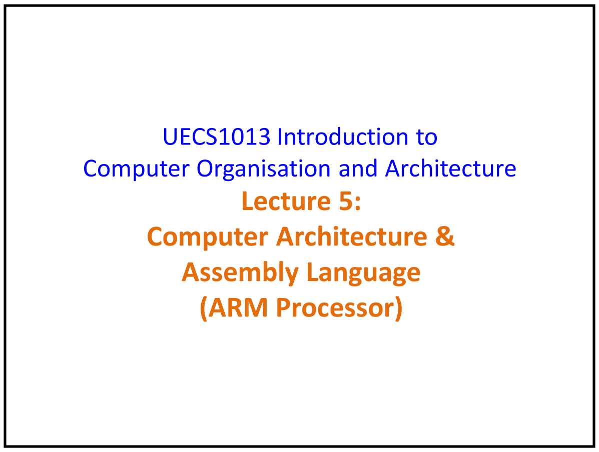
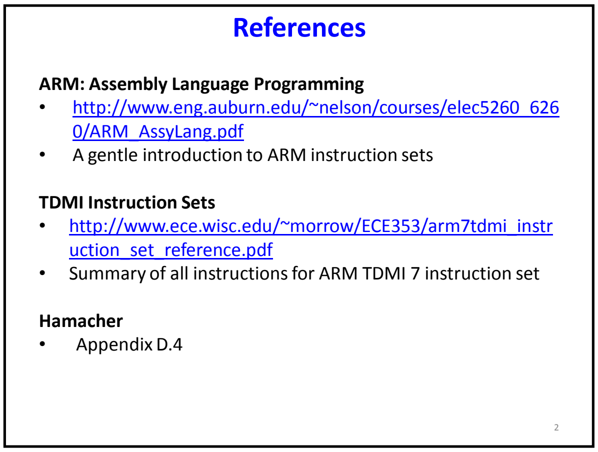


UECS1013 Introduction to
Computer Organisation and Architecture

**Lecture 5:**
**Computer Architecture &**
**Assembly Language**
**(ARM Processor)**



## References

**ARM: Assembly Language Programming**

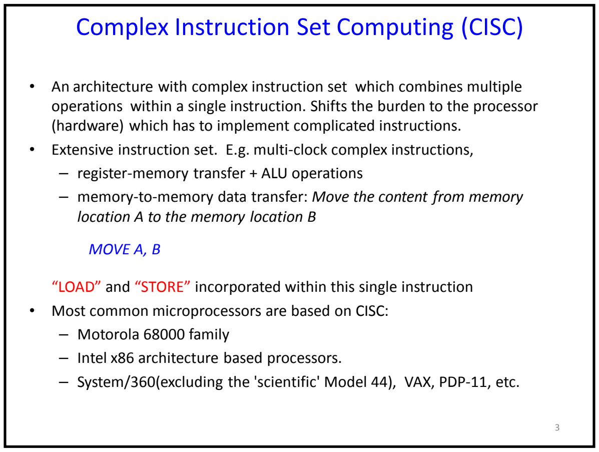
- http://www.eng.auburn.edu/~nelson/courses/elec5260_6260/ARM_AssyLang.pdf
- A gentle introduction to ARM instruction sets

**TDMI Instruction Sets**

- http://www.ece.wisc.edu/~morrow/ECE353/arm7tdmi_instruction_set_reference.pdf
- Summary of all instructions for ARM TDMI 7 instruction set

**Hamacher**

- Appendix D.4

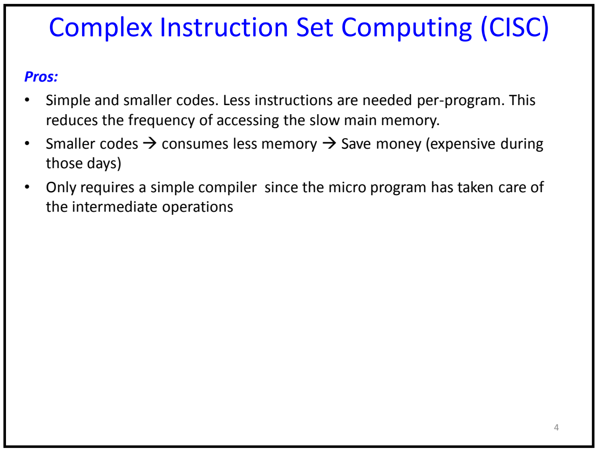


## Complex Instruction Set Computing (CISC)

- An architecture with complex instruction set which combines multiple operations within a single instruction. Shifts the burden to the processor (hardware) which has to implement complicated instructions.
- Extensive instruction set. E.g. multi-clock complex instructions,
  - register-memory transfer + ALU operations
  - memory-to-memory data transfer: *Move the content from memory location A to the memory location B*

    *MOVE A, B*

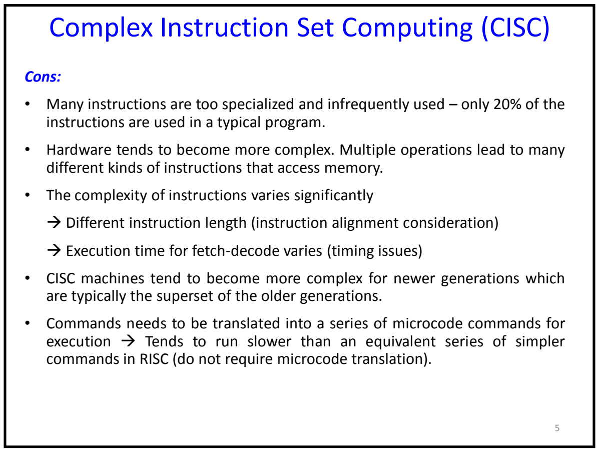
  "LOAD" and "STORE" incorporated within this single instruction
- Most common microprocessors are based on CISC:
  - Motorola 68000 family
  - Intel x86 architecture based processors.
  - System/360(excluding the 'scientific' Model 44), VAX, PDP-11, etc.



## Complex Instruction Set Computing (CISC)

***Pros:***

- Simple and smaller codes. Less instructions are needed per-program. This reduces the frequency of accessing the slow main memory.
- Smaller codes → consumes less memory → Save money (expensive during those days)
- Only requires a simple compiler since the micro program has taken care of the intermediate operations



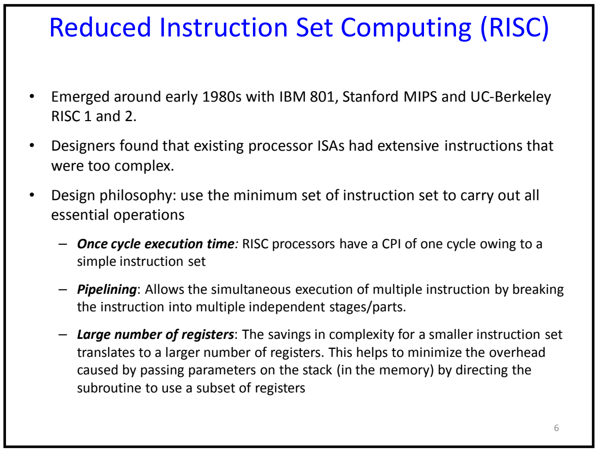
## Complex Instruction Set Computing (CISC)

***Cons:***

- Many instructions are too specialized and infrequently used – only 20% of the instructions are used in a typical program.
- Hardware tends to become more complex. Multiple operations lead to many different kinds of instructions that access memory.
- The complexity of instructions varies significantly

  → Different instruction length (instruction alignment consideration)

  → Execution time for fetch-decode varies (timing issues)
- CISC machines tend to become more complex for newer generations which are typically the superset of the older generations.
- Commands needs to be translated into a series of microcode commands for execution → Tends to run slower than an equivalent series of simpler commands in RISC (do not require microcode translation).



## Reduced Instruction Set Computing (RISC)

- Emerged around early 1980s with IBM 801, Stanford MIPS and UC-Berkeley RISC 1 and 2.
- Designers found that existing processor ISAs had extensive instructions that were too complex.
- Design philosophy: use the minimum set of instruction set to carry out all essential operations
  - ***Once cycle execution time:*** RISC processors have a CPI of one cycle owing to a simple instruction set
  - ***Pipelining***: Allows the simultaneous execution of multiple instruction by breaking the instruction into multiple independent stages/parts.
  - ***Large number of registers***: The savings in complexity for a smaller instruction set translates to a larger number of registers. This helps to minimize the overhead caused by passing parameters on the stack (in the memory) by directing the subroutine to use a subset of registers

## Reduced Instruction Set Computing (RISC)

- *Consistency among instructions* is achieved by
  - Using few simple instructions that are of the same length
  - Memory access is achieved only through explicit data movement commands such as the *load* and *store* instructions
  - Each instruction performs less work but instruction execution time among different instructions is consistent
- The complexity of the operations is moved from ISA to the domain of the assembly programmer/compiler → RISC requires more powerful compiler compared to CISC to translate high-level languages to machine languages
- The codes in RISC are longer than CISC but they run faster.
- Example RISC Processors:

  Apple iPods (custom ARM7TDMI SoC) Apple iPhone (Samsung ARM1176JZF), Palm and PocketPC PDAs and smartphones (Intel XScale family, Samsung SC32442 - ARM9), Nintendo Game Boy Advance (ARM7), Nintendo DS (ARM7, ARM9), Sony Network Walkman (Sony in-house ARM based chip), Some Nokia and Sony Ericsson mobile phones

## CISC versus RISC

CISC RISC

$$CPU\ Time = \frac{\#instructions}{program} \times \frac{average\ \#cycle}{instruction} \times \frac{duration}{cycle}$$

| CISC | RISC |
|---|---|
| • Speed up by reducing #instructions per program | • Speed up by shortening average #cycle/instruction and seconds/cycle |
| • Example:<br>MULT 2:3, 5:2 | • Example:<br>LOAD A, 2:3, LOAD B, 5:2,<br>PROD A, B, STORE 2:3, A |
| • Emphasis on hardware | • Emphasis on software |
| • Memory-to-memory operations are allowed | • Register to register where LOAD and STORE are independent instructions |
| • Transistors used for implementing a huge set of complex instructions | • Less transistors for core, resulting in larger register and cache size. |
| • Multi-clock, non-standardized length instruction | • Single-clock and common-length instructions |

## Hybrid Processors

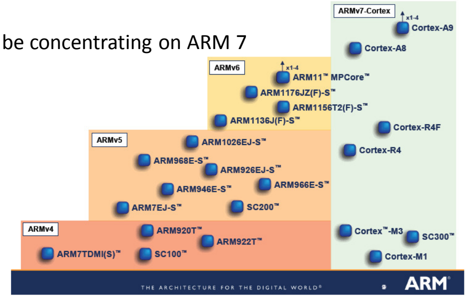
- Recent advances in compiler technology, compiler capability and memory speed and fabrication technology blurs the line between RISC and CISC
- Most modern microprocessor is a hybrid.
  - Modern RISC supports chips as much instructions as yesterday's CISC chips.
  - Many CISC chips use many techniques which are previously only limited to RISC such as pipelining.
  - CISC and RISC systems are becoming more and more alike.

## ARM Processor

- ARM: "Advanced RISC Machine"
- IP Core Provider:
  - Does not manufacture its own VLSI devices
  - Sells license to use its core design to electronic companies such as Apple, Samsung, Philips, ATMEL, Sharp, ST and TI.
- ARM is one of the most widespread processor cores in the world
  - In 2007, 98% of new cell phones uses ARM processor and as of 2009, 90% of all embedded 32-bit processors was an ARM. Recently, Cortex-A is used in IPAD.
- Why ARM?
  - Low power, low cost, tiny: *Suitable for portable devices*
  - *Good performance*

## ARM Processor

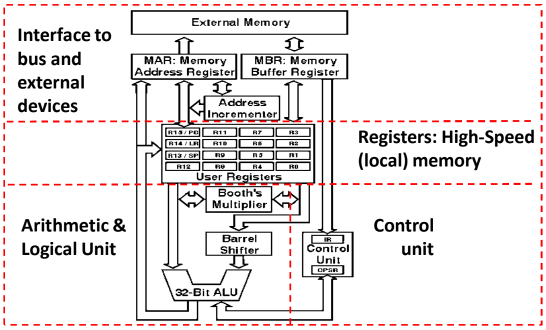
- ARM7 uses ***von Neuman Architecture*** *(single bus for both data and instruction)* while ARM9 and Cortex uses the ***Harvard Architecture*** *(separate data and instruction bus)*.
- We will be concentrating on ARM 7



## ARM Processor

- All instructions are 32 bits long.
- Most instructions execute in a single cycle.
- Every instruction can be conditionally executed.
- RISC-based (load/store) architecture
  - Data processing instructions act only on registers
  - LDR and STR instructions for memory and register data transfer
  - Combined ALU and shifter for high speed bit manipulation
  - Specific memory access instructions with powerful auto-indexing addressing modes.
    - Flexible multiple register load and store instructions

## ARM Processor



Interface to bus and external devices

Registers: High-Speed (local) memory

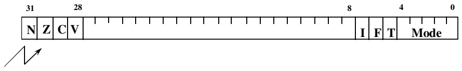
Arithmetic & Logical Unit

Control unit

## Register Organization

- ARM has 37 registers in total, all of which are 32-bits long.
  - 30 ***general purpose registers***
  - 1 dedicated ***program counter (PC)***
  - 1 dedicated ***current program status register (CPSR)***
  - 5 dedicated ***saved program status registers (SPSR)***

## Status Registers (CPSR and SPSR)



Copies of the ALU status flags (latched if the instruction has the "S" bit set).

* **Condition Code Flags**

N = **N**egative result from ALU flag.
Z = **Z**ero result from ALU flag.
C = ALU operation **C**arried out
V = ALU operation o**V**erflowed

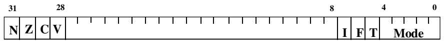
* **Mode Bits**
[4:0] define the processor mode.

* **Interrupt Disable bits.**
**I** = 1, disables the IRQ.
**F** = 1, disables the FIQ.

* **T Bit (Architecture v4T only)**
T = 0, Processor in ARM state
T = 1, Processor in Thumb state

- Holds information of the most recently executed ALU operation ***if suffix "S" is attached to the instruction***.
- Control the enabling and disabling of interrupts.
- Set the processor operating mode

## Status Registers (CPSR and SPSR)



- **N (Negative Flag)**
  - Indicates a negative in the arithmetic operation
  - Example (Subtract 10 from R2)

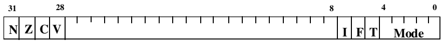
    SUBS R6, R2, R3 ; R2 = 2, R3 = 10

    (CPSR will be updated, N = 1)

    SUB R6, R2, R3 ; R2 = 2, R3 = 10

    (CPSR will **NOT** be updated, N is not updated)

## Status Registers (CPSR and SPSR)

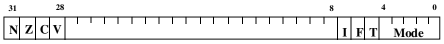


- **Z(Zero Flag)**
  - Indicates a zero in the arithmetic operation
  - Example (Subtract 10 from R2)

    SUBS R6, R2, #10 where R2 = 10
- **C(Carry Flag)**
  - Operations (e.g. addition) results in carry
  - Example:

    ADDS R6, R2, #0xFFFFFFFF where R2 = #0xFFFFFFFF = -1
  - Does not necessarily cause overflow. For example, addition of negative numbers may generate a carry but no overflow.

## Status Registers (CPSR and SPSR)



- **V (Overflow Flag)**
  - An arithmetic overflow has occurred.
  - Example: ADDS R6, R2, #0x7FFFFFFF where R2 = #0x7FFFFFFF

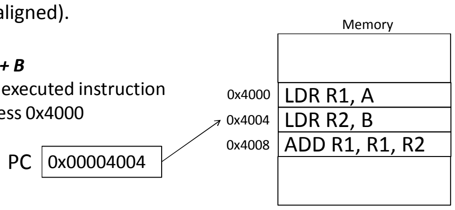
    Two positive number produces a negative number in 2nd Complement!

## Program Counter

- PC is used to point to the next instruction to be executed
- All instructions are 32 bits in length and therefore all instructions must be word aligned
- Therefore the PC value is stored in bits [31:2] with bits [1:0] equal to zero (as instruction cannot be halfword or byte aligned).

***Example: A + B***

- Currently executed instruction is at address 0x4000

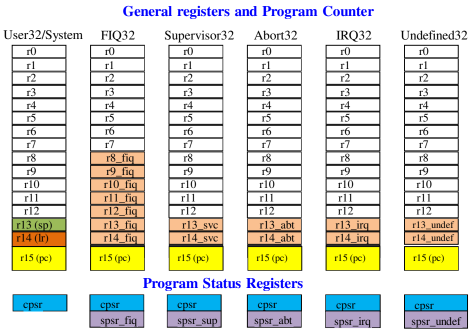


## Operating Modes

- Although there are 37 registers in total, a programmer have access to maximum of 17 registers in user mode and 18 in privileged mode
- The ARM has six different operating modes:
  - ***User*: unprivileged mode under which most tasks run**

  **Privileged Modes:**
  - ***IRQ***: raised when an external device raises a normal interrupt request
  - ***FIQ***: entered when an external device raises a fast-interrupt request to obtain urgent service
  - ***Supervisor***: entered on reset and when a Software Interrupt instruction (SWI) command is executed by a user program
  - ***Abort***: used to handle memory access violations
  - ***Undef***: used to handle undefined instructions

## Operating Modes



**General registers and Program Counter**

| User32/System | FIQ32 | Supervisor32 | Abort32 | IRQ32 | Undefined32 |
|---|---|---|---|---|---|
| r0 | r0 | r0 | r0 | r0 | r0 |
| r1 | r1 | r1 | r1 | r1 | r1 |
| r2 | r2 | r2 | r2 | r2 | r2 |
| r3 | r3 | r3 | r3 | r3 | r3 |
| r4 | r4 | r4 | r4 | r4 | r4 |
| r5 | r5 | r5 | r5 | r5 | r5 |
| r6 | r6 | r6 | r6 | r6 | r6 |
| r7 | r7 | r7 | r7 | r7 | r7 |
| r8 | r8_fiq | r8 | r8 | r8 | r8 |
| r9 | r9_fiq | r9 | r9 | r9 | r9 |
| r10 | r10_fiq | r10 | r10 | r10 | r10 |
| r11 | r11_fiq | r11 | r11 | r11 | r11 |
| r12 | r12_fiq | r12 | r12 | r12 | r12 |
| r13 (sp) | r13_fiq | r13_svc | r13_abt | r13_irq | r13_undef |
| r14 (lr) | r14_fiq | r14_svc | r14_abt | r14_irq | r14_undef |
| r15 (pc) | r15 (pc) | r15 (pc) | r15 (pc) | r15 (pc) | r15 (pc) |

**Program Status Registers**

| | | | | | |
|---|---|---|---|---|---|
| cpsr | cpsr | cpsr | cpsr | cpsr | cpsr |
| | spsr_fiq | spsr_sup | spsr_abt | spsr_irq | spsr_undef |

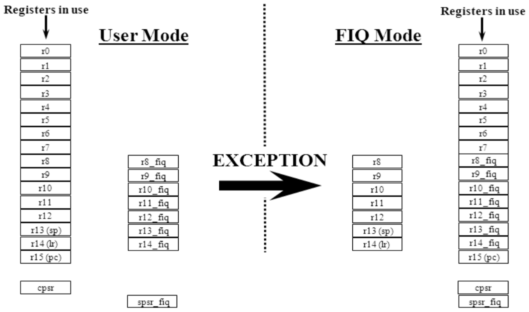
## Operating Modes

- The register files are arranged into several banks. The set of registers which are accessible depends on the ***mode***. Example:
  - R0 to R6 can be accessed in all modes
  - An assembly program running in user mode is allowed access to R8 but a program running in FIQ mode do not, but instead have access to a different register, R8_FIQ.
- Each mode can access
  - a *particular* set of R0-R12 registers.
  - R15 (the ***program counter***)
  - CPSR (the ***current program status register***)
  - a particular r13 (the ***stack pointer***) and r14 (***link register***)

  and privileged modes can also access
  - a particular SPSR (***saved program status register***)

## Operating Modes

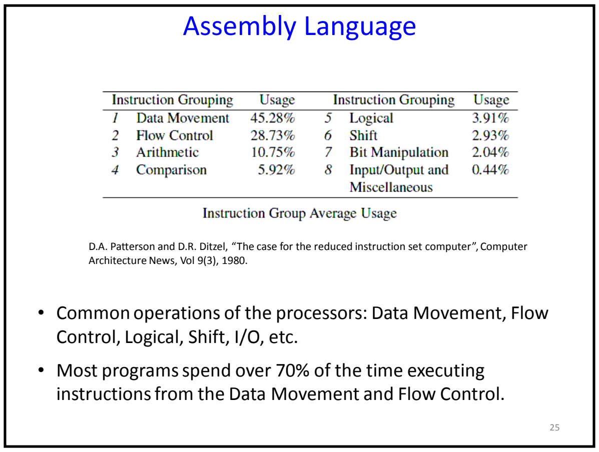
**From User Mode to FIQ Mode**



## Assembly Language

*Why assembly language?*

- Many system-level developer will continue to program in assembly language to have more control over the processor.
- Writing assembly language programs helps to understand computer architecture.
- Many application require the efficiency of assembly language
- Learning assembly language helps to know what is happening inside the computer.
- The foundation of many abstract concepts in high-level programming languages and operating systems etc. closely related to assembly language and computer architecture.



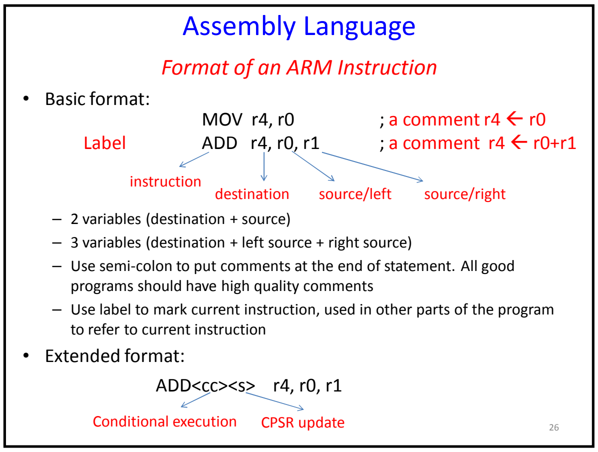
## Assembly Language

| Instruction Grouping | | Usage | Instruction Grouping | | Usage |
|---|---|---|---|---|---|
| *1* | Data Movement | 45.28% | *5* | Logical | 3.91% |
| *2* | Flow Control | 28.73% | *6* | Shift | 2.93% |
| *3* | Arithmetic | 10.75% | *7* | Bit Manipulation | 2.04% |
| *4* | Comparison | 5.92% | *8* | Input/Output and Miscellaneous | 0.44% |

Instruction Group Average Usage

D.A. Patterson and D.R. Ditzel, "The case for the reduced instruction set computer", Computer Architecture News, Vol 9(3), 1980.

- Common operations of the processors: Data Movement, Flow Control, Logical, Shift, I/O, etc.
- Most programs spend over 70% of the time executing instructions from the Data Movement and Flow Control.



## Assembly Language

*Format of an ARM Instruction*

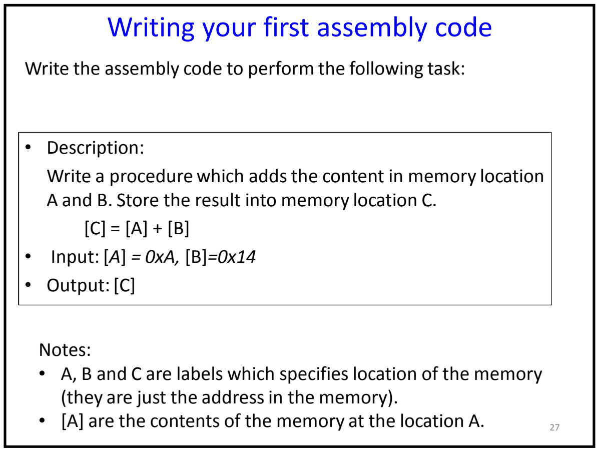
- Basic format:

```
                MOV  r4, r0             ; a comment r4 ← r0
Label           ADD  r4, r0, r1         ; a comment  r4 ← r0+r1
```

instruction
destination
source/left
source/right

  - 2 variables (destination + source)
  - 3 variables (destination + left source + right source)
  - Use semi-colon to put comments at the end of statement. All good programs should have high quality comments
  - Use label to mark current instruction, used in other parts of the program to refer to current instruction
- Extended format:

```
ADD<cc><s>   r4, r0, r1
```



## Writing your first assembly code

Write the assembly code to perform the following task:

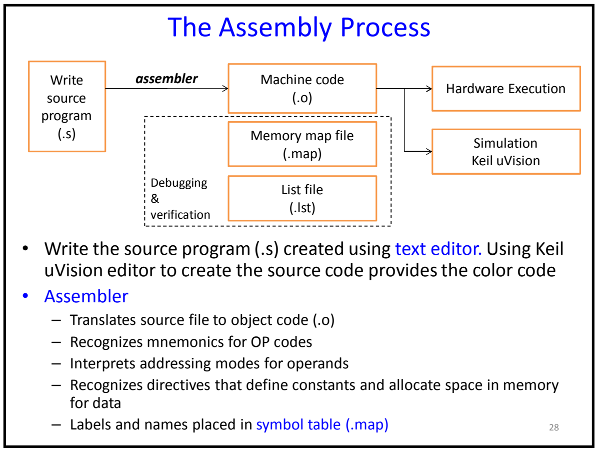
- Description:

  Write a procedure which adds the content in memory location A and B. Store the result into memory location C.

  [C] = [A] + [B]
- Input: [*A*] = *0xA*, [B]=*0x14*
- Output: [C]

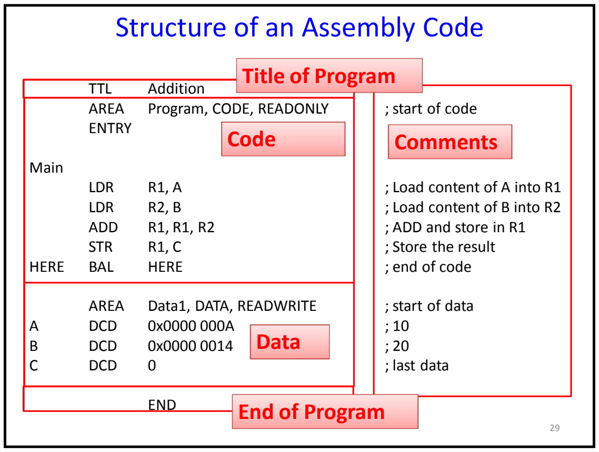
Notes:

- A, B and C are labels which specifies location of the memory (they are just the address in the memory).
- [A] are the contents of the memory at the location A.



## The Assembly Process

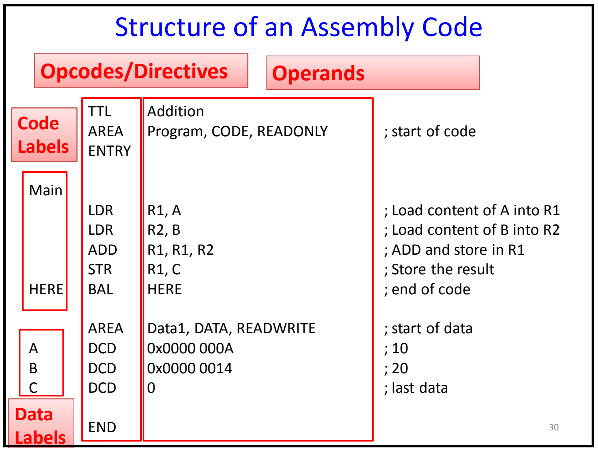
- Write the source program (.s) created using text editor. Using Keil uVision editor to create the source code provides the color code
- Assembler
  - Translates source file to object code (.o)
  - Recognizes mnemonics for OP codes
  - Interprets addressing modes for operands
  - Recognizes directives that define constants and allocate space in memory for data
  - Labels and names placed in symbol table (.map)



## Structure of an Assembly Code

```
        TTL     Addition
        AREA    Program, CODE, READONLY        ; start of code
        ENTRY
Main
        LDR     R1, A                          ; Load content of A into R1
        LDR     R2, B                          ; Load content of B into R2
        ADD     R1, R1, R2                     ; ADD and store in R1
        STR     R1, C                          ; Store the result
HERE    BAL     HERE                           ; end of code

        AREA    Data1, DATA, READWRITE         ; start of data
A       DCD     0x0000 000A                    ; 10
B       DCD     0x0000 0014                    ; 20
C       DCD     0                              ; last data

                END
```

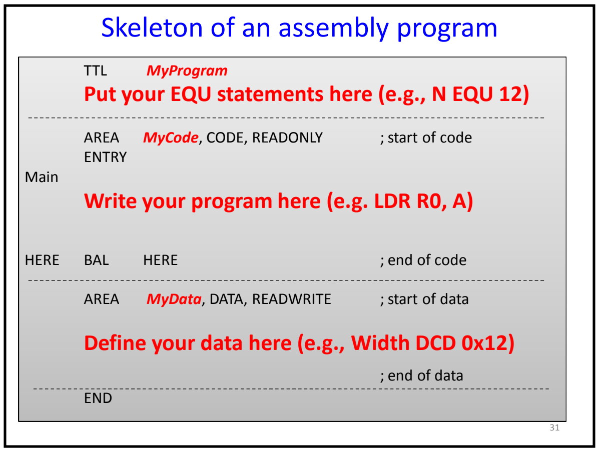
Title of Program
Code
Comments
Data
End of Program



## Structure of an Assembly Code

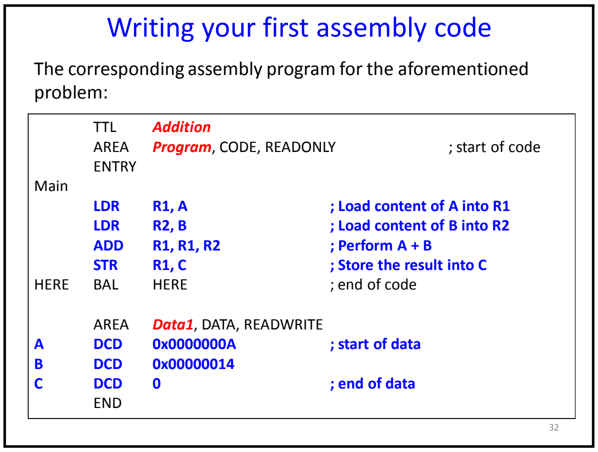
Opcodes/Directives
Operands
Code Labels
Data Labels

```
        TTL     Addition
        AREA    Program, CODE, READONLY        ; start of code
        ENTRY

Main
        LDR     R1, A                          ; Load content of A into R1
        LDR     R2, B                          ; Load content of B into R2
        ADD     R1, R1, R2                     ; ADD and store in R1
        STR     R1, C                          ; Store the result
HERE    BAL     HERE                           ; end of code

        AREA    Data1, DATA, READWRITE         ; start of data
A       DCD     0x0000 000A                    ; 10
B       DCD     0x0000 0014                    ; 20
C       DCD     0                              ; last data

        END
```



## Skeleton of an assembly program

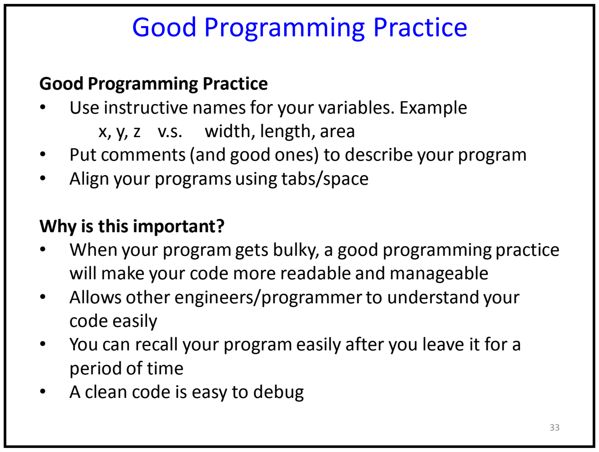
```
        TTL     MyProgram
        Put your EQU statements here (e.g., N EQU 12)
------------------------------------------------------------------
        AREA    MyCode, CODE, READONLY          ; start of code
        ENTRY
Main
        Write your program here (e.g. LDR R0, A)

HERE    BAL     HERE                            ; end of code
------------------------------------------------------------------
        AREA    MyData, DATA, READWRITE         ; start of data

        Define your data here (e.g., Width DCD 0x12)
                                                ; end of data
------------------------------------------------------------------
        END
```



## Writing your first assembly code

The corresponding assembly program for the aforementioned problem:

```
        TTL     Addition
        AREA    Program, CODE, READONLY         ; start of code
        ENTRY
Main
        LDR     R1, A                   ; Load content of A into R1
        LDR     R2, B                   ; Load content of B into R2
        ADD     R1, R1, R2              ; Perform A + B
        STR     R1, C                   ; Store the result into C
HERE    BAL     HERE                    ; end of code

        AREA    Data1, DATA, READWRITE
A       DCD     0x0000000A              ; start of data
B       DCD     0x00000014
C       DCD     0                       ; end of data
        END
```



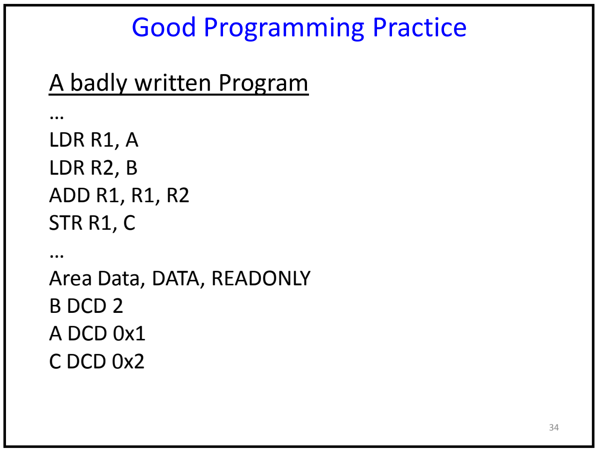
## Good Programming Practice

**Good Programming Practice**

- Use instructive names for your variables. Example
  x, y, z v.s. width, length, area
- Put comments (and good ones) to describe your program
- Align your programs using tabs/space

**Why is this important?**

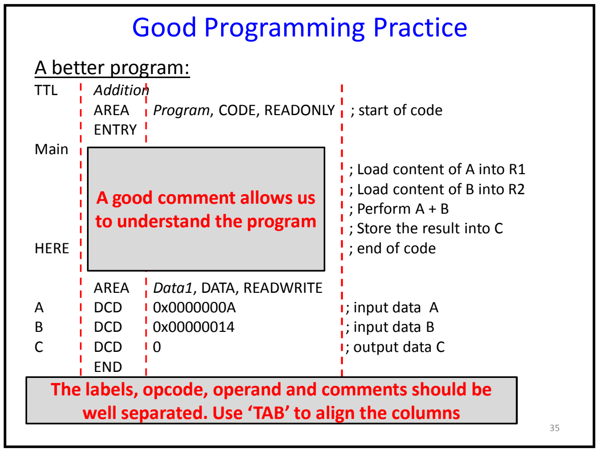
- When your program gets bulky, a good programming practice will make your code more readable and manageable
- Allows other engineers/programmer to understand your code easily
- You can recall your program easily after you leave it for a period of time
- A clean code is easy to debug



## Good Programming Practice

A badly written Program

```
…
LDR R1, A
LDR R2, B
ADD R1, R1, R2
STR R1, C
…
Area Data, DATA, READONLY
B DCD 2
A DCD 0x1
C DCD 0x2
```



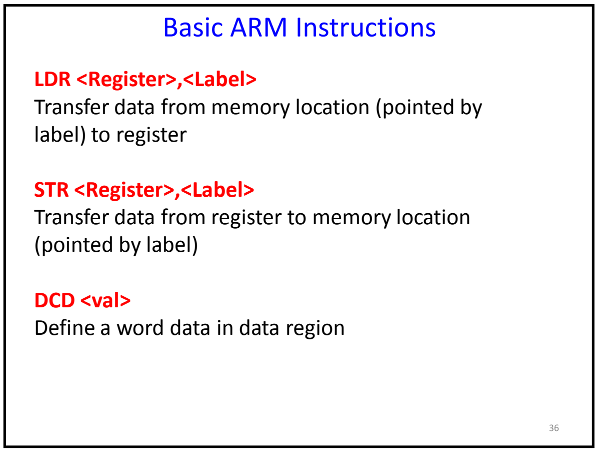
## Good Programming Practice

A better program:

```
TTL   | Addition
      | AREA  | Program, CODE, READONLY | ; start of code
      | ENTRY |
Main  |
      |                                 | ; Load content of A into R1
      |                                 | ; Load content of B into R2
      |                                 | ; Perform A + B
      |                                 | ; Store the result into C
HERE  |                                 | ; end of code

      | AREA  | Data1, DATA, READWRITE  |
A     | DCD   | 0x0000000A              | ; input data  A
B     | DCD   | 0x00000014              | ; input data B
C     | DCD   | 0                       | ; output data C
      | END   |
```

A good comment allows us to understand the program

The labels, opcode, operand and comments should be well separated. Use 'TAB' to align the columns



## Basic ARM Instructions

**LDR <Register>,<Label>**

Transfer data from memory location (pointed by label) to register

**STR <Register>,<Label>**

Transfer data from register to memory location (pointed by label)

**DCD <val>**

Define a word data in data region

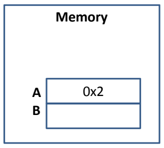
## Example 1

Write the assembly code to perform the following task:

- Description:

  Write a procedure which copies the value from memory location in A to B. Store

  [B] = [A]
- Input: [*A*]=0x2
- Output: [B]



## Example 1

```
        TTL     CopyOneItem
        AREA    MyFirstProgram, CODE, READONLY

        ENTRY
Main
        LDR     R1, A           ; R1 ← [A]
        STR     R1, B           ; [B] ← R1
HERE    BAL     HERE            ; end of code

        AREA    MyData, DATA, READWRITE
A       DCD     0x2             ; input (hexadecimal)
B       DCD     0               ; output (initialize to 0)
        END
```

## Basic ARM Instructions

**MUL <Rdest>, <Rsource1>,<Rsource2>**
Rdest = Rsource 1 x Rsource2

**ADD <Rdest>, <Rsource1>, <Rsource2>**
Rdest = Rsource 1 + Rsource2

**ADD <Rdest>, <Rsource1>, #<Value>**
Rdest = Rsource 1 + *Value*

**SUB <Rdest>, <Rsource1>, <Rsource2>**
Rdest = Rsource 1 - Rsource2

**SUB <Rdest>, <Rsource1>, #<Value>**
Rdest = Rsource 1 - *Value*

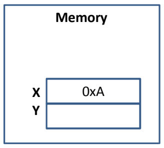
## Example 2

Write the assembly code to perform the following task:

- Description:

  Write a procedure which performs the following operation

  $y = x^2 + 4$
- Input: [X] = 10
- Output: [Y]



## Example 2

```
        TTL     Func1
        AREA    MyProgram, CODE, READONLY

        ENTRY
Main
        LDR     R1, X           ; R1 ← [X]
        MUL     R2, R1, R1      ; R2 ← X²
        ADD     R2, R2, #4      ; R2 ← X² + 4
        STR     R2, Y           ; [Y] ← R2 (result)
HERE    BAL     HERE            ; end of code

        AREA    MyData , DATA, READWRITE
X       DCD     10              ; input (decimal)
Y       DCD     0               ; output (initialize to 0)
        END
```

## Basic ARM Instructions

**MOV <Rdest>, #value**
Update Rdest with value

**MOV <Rdest>,<Rsource>**
Transfer data from Rsource to Rdest

**LDR <Rdest>, =<Label>**
Load Rdest with the memory address that label represents (not the content pointed by label)

**LDR <Rdest>, [<Rsource>]**
Load Rdest with the memory address that pointed by Rsource

**STR <Rsource>, [<Rdest>]**
Store Rdest into the memory address pointed by Rdest

## Example 3:

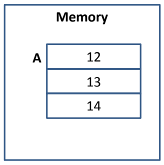
Write the assembly code to perform the following task:

- Description:

  Write a procedure which resets a three-item array in memory location A to 0.

  Input: [*A*]=12, 13, 14

  Output: [A] = 0, 0, 0



## Example 3

```
        TTL     ResetArray
        AREA    Program, CODE, READONLY
        ENTRY
Main
        MOV     R0, #0
        LDR     R1, =A          ; R1 = A (first item in array A)
        STR     R0, [R1]        ; Set [A] = 0
        ADD     R1, R1, #4      ; R1 = A + 4 (second item)
        STR     R0, [R1 ]       ; Set [A + 4] = 0
        ADD     R1, R1, #4      ; R1 = A + 8 (third item)
        STR     R0, [R1]        ; Set [A + 8] = 0
HERE    BAL     HERE                    ; end of code

        AREA    Data1, DATA, READWRITE
A       DCD     12, 13,14       ; array A (2 items)
B       DCD     0, 0            ; array B (2 items)
        END
```

## Example 4

Write the assembly code to perform the following task:

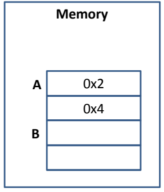
- Description:

  Write a procedure which copies a two-items array from memory location in A to B.

  [B] = [A]

  [B+4] = [A+4]

- Input: [*A*]=0x2, [A+4]=0x4
- Output: [B], [B+4]



## Example 4

```
        TTL     CopyTwoItems
        AREA    Program, CODE, READONLY         ; start of code
        ENTRY
Main
        LDR     R1, =A          ; R1 = A (first item in array A)
        LDR     R2, =B          ; R2 = B (first item in array B)
        LDR     R3, [R1]        ; Get [A]
        STR     R3, [R2]        ; [B] ← [A]
        ADD     R1, R1, #4      ; R1 = A + 4 (second item in array A)
        ADD     R2, R2, #4      ; R2 = B + 4 (second item in array B)
        LDR     R3, [R1]        ; Get [A+4]
        STR     R3, [R2]        ; [B+4] ← [A+4]
HERE    BAL     HERE                    ; end of code

        AREA    Data1, DATA, READWRITE
A       DCD     0x2, 0x4        ; array A (2 items)
B       DCD     0, 0            ; array B (2 items)
        END
```

## CONDITIONAL OPERATION

## Conditional Update of Status Register

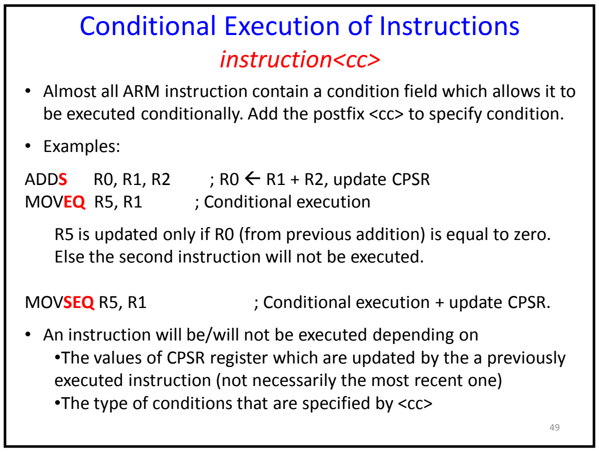
### *instruction<s>*

- The processor execute the instructions pointed by Program Counter. It will *additionally* update the CPSR register if the instruction is appended with <s> :
  - N (negative), Z (zero), C(carry), V(overflow)
- Example:

```
MOVS R0, R1         ; will set the N and Z flags
                    ; depending on the value of R1
MOV R0, R1          ; CPSR will NOT be updated
```

- The CPSR is updated depending on the result of the operation. For example:

```
ADDS R1, R2, R3
if  R2 = -1, R3 = 1, then NZCV = 0110         ; zero and carry set
if R2 = 10, R3 = -20, then NZCV = 1010 ; negative and carry set
```



## Conditional Execution of Instructions

*instruction<cc>*

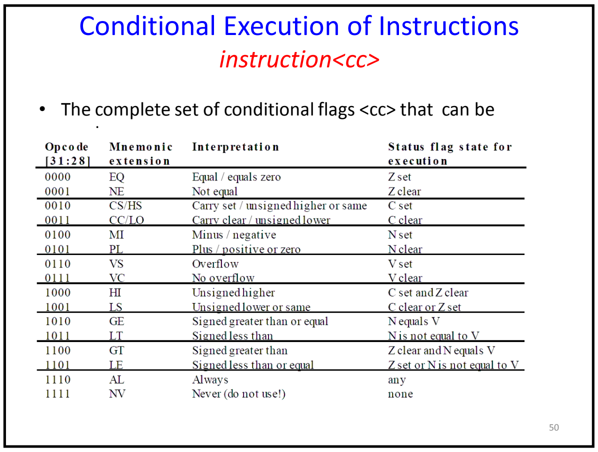
- Almost all ARM instruction contain a condition field which allows it to be executed conditionally. Add the postfix <cc> to specify condition.
- Examples:

```
ADDS     R0, R1, R2       ; R0 ← R1 + R2, update CPSR
MOVEQ  R5, R1          ; Conditional execution
```

R5 is updated only if R0 (from previous addition) is equal to zero. Else the second instruction will not be executed.

```
MOVSEQ R5, R1                 ; Conditional execution + update CPSR.
```

- An instruction will be/will not be executed depending on
  - •The values of CPSR register which are updated by the a previously executed instruction (not necessarily the most recent one)
  - •The type of conditions that are specified by <cc>

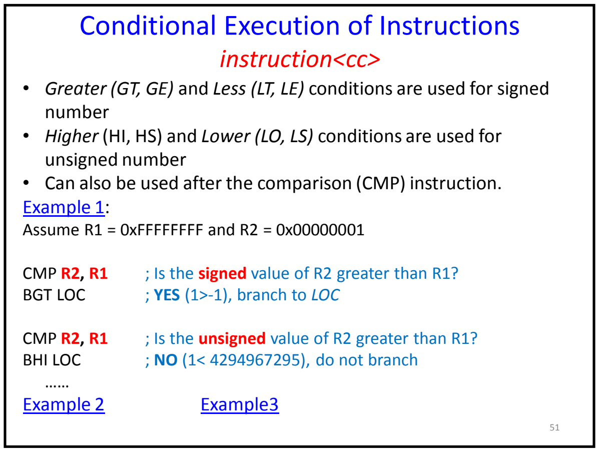


## Conditional Execution of Instructions

*instruction<cc>*

- The complete set of conditional flags <cc> that can be

| Opcode [31:28] | Mnemonic extension | Interpretation | Status flag state for execution |
|---|---|---|---|
| 0000 | EQ | Equal / equals zero | Z set |
| 0001 | NE | Not equal | Z clear |
| 0010 | CS/HS | Carry set / unsigned higher or same | C set |
| 0011 | CC/LO | Carry clear / unsigned lower | C clear |
| 0100 | MI | Minus / negative | N set |
| 0101 | PL | Plus / positive or zero | N clear |
| 0110 | VS | Overflow | V set |
| 0111 | VC | No overflow | V clear |
| 1000 | HI | Unsigned higher | C set and Z clear |
| 1001 | LS | Unsigned lower or same | C clear or Z set |
| 1010 | GE | Signed greater than or equal | N equals V |
| 1011 | LT | Signed less than | N is not equal to V |
| 1100 | GT | Signed greater than | Z clear and N equals V |
| 1101 | LE | Signed less than or equal | Z set or N is not equal to V |
| 1110 | AL | Always | any |
| 1111 | NV | Never (do not use!) | none |



## Conditional Execution of Instructions

*instruction<cc>*

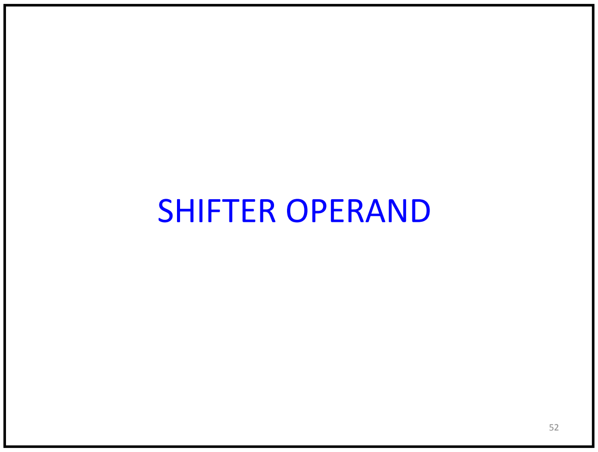
- *Greater (GT, GE)* and *Less (LT, LE)* conditions are used for signed number
- *Higher* (HI, HS) and *Lower (LO, LS)* conditions are used for unsigned number
- Can also be used after the comparison (CMP) instruction.

Example 1:

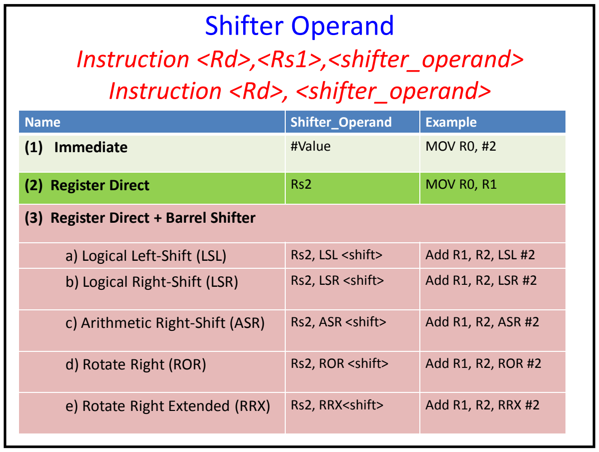
Assume R1 = 0xFFFFFFFF and R2 = 0x00000001

```
CMP R2, R1        ; Is the signed value of R2 greater than R1?
BGT LOC           ; YES (1>-1), branch to LOC

CMP R2, R1        ; Is the unsigned value of R2 greater than R1?
BHI LOC           ; NO (1< 4294967295), do not branch
    ……
```

Example 2                    Example3



## SHIFTER OPERAND

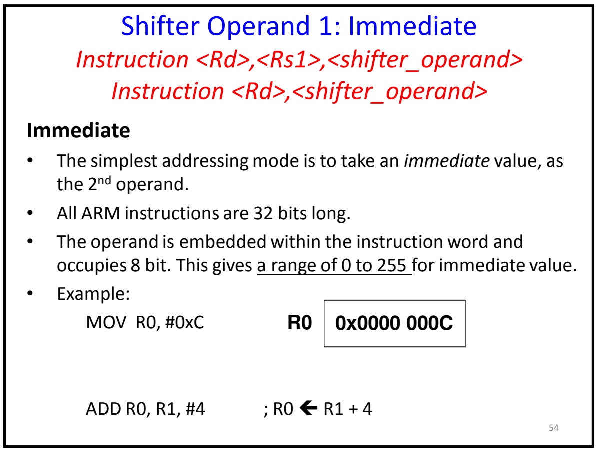


## Shifter Operand

*Instruction <Rd>,<Rs1>,<shifter_operand>*

*Instruction <Rd>, <shifter_operand>*

| Name | Shifter_Operand | Example |
|---|---|---|
| **(1) Immediate** | #Value | MOV R0, #2 |
| **(2) Register Direct** | Rs2 | MOV R0, R1 |
| **(3) Register Direct + Barrel Shifter** | | |
| a) Logical Left-Shift (LSL) | Rs2, LSL <shift> | Add R1, R2, LSL #2 |
| b) Logical Right-Shift (LSR) | Rs2, LSR <shift> | Add R1, R2, LSR #2 |
| c) Arithmetic Right-Shift (ASR) | Rs2, ASR <shift> | Add R1, R2, ASR #2 |
| d) Rotate Right (ROR) | Rs2, ROR <shift> | Add R1, R2, ROR #2 |
| e) Rotate Right Extended (RRX) | Rs2, RRX<shift> | Add R1, R2, RRX #2 |



## Shifter Operand 1: Immediate

*Instruction <Rd>,<Rs1>,<shifter_operand>*

*Instruction <Rd>,<shifter_operand>*

**Immediate**

- The simplest addressing mode is to take an *immediate* value, as the 2nd operand.
- All ARM instructions are 32 bits long.
- The operand is embedded within the instruction word and occupies 8 bit. This gives a range of 0 to 255 for immediate value.
- Example:

```
MOV  R0, #0xC              R0   0x0000 000C


ADD R0, R1, #4           ; R0 ← R1 + 4
```

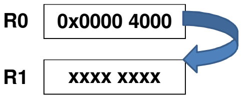
## Shifter Operand 2: Register Direct

*Instruction <Rd>,<Rs1(Opt)>,<shifter_operand>*

**Register Direct**

- The *Register Direct* addressing mode transfers data between registers
- Example: Transfers register content from R0 to R1

MOV R1, R0

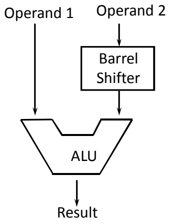


ADD R1, R2, R3 ; R1 ← R2 + R3

## Shifter Operand 3: Register Direct + Barrel Shifter

*Instruction <Rd>,<Rs1 (Opt)>,<shifter_operand>,**<shift>***

**Register Direct + Barrel Shifter**



- *Operand 2* can also be taken from a ***register*** which can be optionally applied with ***shift*** operation.
  Example: MOV, R0, R1, LSL #4
  R0: Operand 1
  R1: Operand 2
  #4: Shift value
- The shift value can be either be:
  - 5 bit unsigned integer: *Maximum shift of 32 positions*
  - Specified in the bottom byte of another register.

## Shifter Operand 3: Register Direct + Barrel Shifter

*Instruction <Rd>,<Rs1 (Opt)>,<shifter_operand>,**<shift>***

- The shift operation allows the operand to undergo some pre-processing before performing the actual arithmetic
- Example:
  MOV, R0, R1, LSL #4 ; assume content of R1 = 0x12345678

  1. Get the value of R1 (**<u>R1 remains untouched</u>**)
     Original input = 0x12345678
  2. Shift the value left by 4 bits
     Processed input = 0x23456780
  3. Move the final value to R0
     R0 = 0x23456780
     R1 = 0x12345678

## Shifter Operand 3: Register Direct + Barrel Shifter

*Instruction <Rd>,<Rs1 (Opt)>,<shifter_operand>,**<shift>***

*Does the using the barrer shifter incurs any additional cost?*

- When the **<u>immediate value</u>** of 5-bits field (Range = 0 to 31) is used for barrel shift
  *MOV R0, R1, ROR #4*
  - No performance overhead incurred
  - Shift is done for free - executes in single cycle.
- When the bottom byte of a **<u>register</u>** (not PC) is used
  *MOV R0, R1, ROR R2*
  - Then takes extra cycle to execute
  - ARM doesn't have enough read ports to read 3 registers at once.
  - Then, the performance is reduced to be similar to other processors where shift is separate instruction

## Shifter Operand 3: Register Direct + Barrel Shifter

*Instruction <Rd>,<Rs1 (Opt)>,<shifter_operand>,**<shift>***

*What is the purpose of having the Barrel Shifter for Operand 2?*

- It's free (not incurring any extra cost when properly used)
- *Shift* is a common operation for many programs, e.g. DSP algorithms.
- Multiplication instruction is typically much slower than addition in practice

  Example: r0 = r1 * 13
  = r1 *(1 + 4 + 8)
  = r1 + r1*4 + r1*8
  = r1 + Shift_Left (r1, 2) + Shift_Left (r1, 3)

  ADD r0, r1, **r1, LSL #2** ; r0 = r1 + shift(r1, 2)
  ADD r0, r0, **r1, LSL #3** ; r0 = r0 + shift(r1, 3)

## Shifter Operand 3: Register Direct + Barrel Shifter

*Instruction <Rd>,<Rs1 (Opt)>,<shifter_operand>,**<shift>***

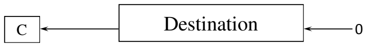
- The ARM doesn't have actual shift instructions.
- Instead it has a barrel shifter which provides a mechanism to carry out shifts as part of other instructions
- There are five types of barrel-shift operations:
  - left shift (*LSL*)
  - logical right-shift (*LSR*)
  - arithmetic right-shift (*ASR*)
  - Rotate right (*ROR*)
  - Rotate right extended (*RRX*)
- LSL, LSR, ASR and ROR are pseudo-instructions (not stand-alone) which are inserted as part of another primary instruction

## Shifter Operand 3: Register Direct + Barrel Shifter

*Instruction <Rd>,<Rs1 (Opt)>,<shifter_operand>,**<shift>***

***Logical Shift Left (LSL)***

•Shifts left by the specified amount. The lower bits of the value are appended by 0. Value of shifter_operand remains unchanged



- Example:

**Assume that R1 = 0x0000 0003, R2 = 0x0000 0005**

| | |
|---|---|
| ***MOV R0, R1, LSL #2*** | R0 = 0x0000 000C (3x4 = 12)<br>R1 = 0x0000 0003 (unchanged) |
| ***MOV R0, R1, LSL R2*** | R0 = 0x0000 0060 (3x32 = 96)<br>R1 = 0x0000 0003 (unchanged) |

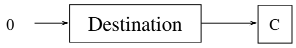
- This is a simple way of performing a ***multiply*** by a power of 2 (x$2^n$). For example, LSL #5 = multiply by 32

## Shifter Operand 3: Register Direct + Barrel Shifter

*Instruction <Rd>,<Rs1 (Opt)>,<shifter_operand>,**<shift>***

***Logical Shift Right (LSR)***

- Shifts left by the specified amount. The lower bits of the value are appended by 0. Value of operand two remains unchanged



- Example: Assume that R1 = **0x0000 000C**, R2 = 0x0000 0003

| | |
|---|---|
| ***MOV R0, R1, LSR #2*** | R0 = 0x0000 0003<br>R1 = 0x0000 000C (unchanged) |
| ***MOV R0, R1, LSR R2*** | R0 = 0x0000 0001<br>R1 = 0x0000 000C (unchanged) |

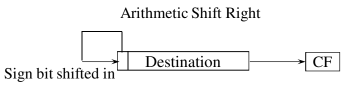
- This is a simple way of performing an ***unsigned divide*** by a power of 2 (|÷$2^n$|). For example, LSL #5 = divide by 32

## Shifter Operand 3: Register Direct + Barrel Shifter

*Instruction <Rd>,<Rs1 (Opt)>,<shifter_operand>,**<shift>***

***Arithmetic Shift Right (ASR)***

- Shifts right (**signed divide** by powers of 2) by the specified amount *& preserves the sign bit* for 2's complement operations. E.g. ASR #5 = div by 32



- Example:

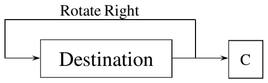
| | |
|---|---|
| ***MOV R0, R1, ASR #2***<br>or<br>***MOV R0, R1, ASR R2***<br>where R2 = 2 | If R1 = 0x00000010 → R0 = 0x0000 0004<br>if R1 = 0xFFFF FFF0 → R0 = 0xFFFF FFFC<br>R1 remains unchanged |

## Shifter Operand (3.4)

*Instruction <Rd>,<Rs1 (Opt)>,<shifter_operand>,**<shift>***

***Rotate Right (ROR)***

- Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB. The last bit to be shifted out is also used as the carry out



- Example: Assume that R1 = **0x8000 0002**, R2 = 0x0000 0003

| | |
|---|---|
| ***MOV R0, R1, LSL #2*** | R0 = 0xA000 0000<br>R1 = 0x8000 0002 (unchanged) |
| ***MOV R0, R1, LSL R2*** | R0 = 0x5000 0000<br>R1 = 0x8000 0002(unchanged) |

## Example 1

Write the assembly code to perform the following task:

- Description:

  Write a procedure which performs the following operation

  Y = 20*X1 + X2
- Input: M[X1] = 2, M[X2] = 3
- Output: M[Y]

But MUL is slow!!!!

```
        TTL     func1_slow
        AREA    MyProgram, CODE, READONLY          ; start of code
        ENTRY
Main
        LDR     R0, X1          ; Load content of X1 into R0
        LDR     R1, X2          ; Load content of X2 into R1
        MOV     R2, #20         ; R2 = 20 (for multiplication)
        MUL     R3, R0, R2      ; R3 = 20X1 (Register Direct)
        ADD     R4, R3, R1      ; R4 = 20X1 + X2  (Immediate)
        STR     R4, Y           ; Save result into Y
HERE    BAL     HERE            ; end of code

        AREA    Data1, DATA, READWRITE
X1      DCD     2                       ; input 1
X2      DCD     3                       ; input 2
Y       DCD     0                       ; output
        END
```

Alternative way:

- The instruction “MUL R3, R0, #20” or 20X1 can be converted into a series of shift operations as follows

R3 = R0 * 20

= R0 *(4 + 16)

= R0*4 + R0*16

= LSL(R0, 2) + LSL (R0, 4)

- Instructions:

**MOV R2, #20** → **MOV R3, R0, LSL #2**

**MUL R3, R0, R2** **ADD R3, R3, R0, LSL #4**

```
        TTL     func1_fast
        AREA    MyProgram, CODE, READONLY       ; start of code
        ENTRY
Main
        LDR     R0, X1                  ; Load content of X1 into R0
        LDR     R1, X2                  ; Load content of X2 into R1
        MOV     R3, R0, LSL #2          ; 4*X1 (Barrel Shift)
        ADD      R3, R3, R0, LSL #4     ; 4*X1 + 16*X1 ( Barrel Shift)
        ADD     R3, R3, R1              ; 20X1 + X2 (Register Direct)
        STR     R3, Y                   ; Save result into Y
HERE    BAL     HERE                    ; end of code

        AREA    Data1, DATA, READWRITE
X1      DCD     2                       ; input 1
X2      DCD     3                       ; input 2
Y       DCD     0                       ; output
        END
```

## Example 2

Write the assembly code to perform the following task:

- Description:

  Write a procedure which performs the second order complement of -X

  Y = 2ndComplement (X)

- Input: M[X] = 0x00010001
- Output: M[Y]

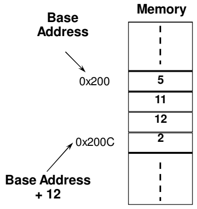
```
        TTL     secondcomp
        AREA    MyProgram, CODE, READONLY       ; start of code
        ENTRY
Main
        LDR     R0, X                   ; Load content of X into R0
        MVN     R0, R0                  ; Invert all bits in R0
        ADD     R0, R0, #1              ; Add 1 to inverted bits
        STR     R0, Y                   ; Store result
        HERE    BAL     HERE            ; end of code

        AREA    Data1, DATA, READWRITE
X       DCD     0x00010001              ; input
Y       DCD     0                       ; output
        END
```

# ADDRESSING MODES

## Addressing Modes

*LDR/STR <Rd>,<addressing_mode>*

- Programs use data structures to organize the information used in computations
- One common example is a table containing *N* integers which are stored in a contiguous memory segment
- *How to get the fourth integer from the table?*
  - Provide the absolute address → not flexible
  - Provide the base address of the table + offset → Need only to change the base address if table is loaded elsewhere in the memory
- Addressing modes provide different ways to specify the operand locations w.r.t. a base address



*****Only applicable to LDR and STR**

# Addressing Modes

*LDR/STR <Rd>,<addressing_mode>*

| Name | Syntax | Addressing Function | Example |
|---|---|---|---|
| **Register indirect** | [Ri] | EA = [Ri] | LDR R0, [R1] |
| **Pre-Indexed**<br>**Pre-indexed with WriteBack**<br>**Post-indexed**<br>**Relative** | [Ri, X]<br>[Ri, X]!<br>[Ri], X<br>*Label* | EA = [Ri] + X; Rk unchanged<br>EA = [Ri]+ X, Rk ← Rk + X<br>EA = [Ri]; Rk ← Rk + X<br>EA = [PC] + offset | LDR R0, [R1, #4]<br>LDR R0, [R1, #4]!<br>LDR R0, [R1], #4<br>BRA *Label* |

EA = Effective Address
X = index/offset value
Label = a label pointing to the memory location

# Register Indirect

*LDR/STR <Rd>,<addressing_mode>*

***Register Indirect***

- *Immediate* and *Register* addressing mode does not handle memory access.
- Data transfer between the system memory and a register in the CPU is **handled by the *LDR* and *STR* instructions only.**
- The *indirect* mode is used to access the location of the memory. The location to be accessed is stored in a register where the register plays the role of a *pointer*.
- The *Indirect* mode provides address in register:

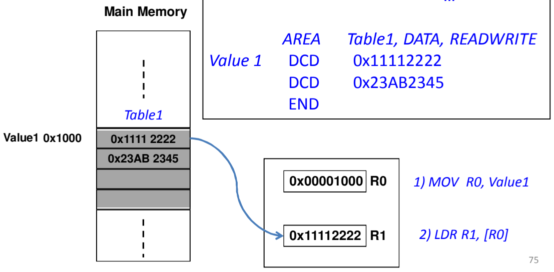
```
LDR      R2, [R5]      ; R2 ← M[R5]
```

# Register Indirect

*LDR/STR <Rd>,<addressing_mode>*

***Register Indirect***

```
            AREA    Code,  CODE, READONLY
            LDR     R0, =Value1
            LDR     R1, [R0]
                          ...

            AREA    Table1, DATA, READWRITE
Value 1     DCD     0x11112222
            DCD     0x23AB2345
            END
```



# Register Indirect + Indexed

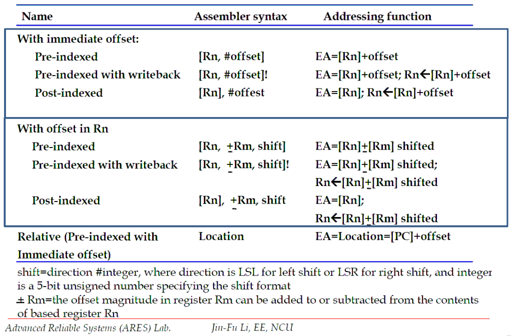
*LDR/STR <Rd>,<addressing_mode>*

***Register Indirect + Indexed***

- ***Pre-indexed mode***: LDR R0, [R1, *Offset*]
  The ***effective address*** of the operand is the sum of the contents of the base register Rn and an offset value
- ***Pre-indexed with writeback mode***: LDR R0, [R1, *Offset*]!
  The effective address of the operand is generated in the same way as in the Pre-indexed mode, and then the effective address is written back into Rn
- ***Post-indexed mode***: : LDR R0, [R1], *Offset*
  The effective address of the operand is the contents of Rn. The offset is then added to this address and the result is written back into Rn

# Register Indirect + Indexed

*LDR/STR <Rd>,<addressing_mode>*



| Name | Assembler syntax | Addressing function |
|---|---|---|
| With immediate offset: | | |
| Pre-indexed | [Rn, #offset] | EA=[Rn]+offset |
| Pre-indexed with writeback | [Rn, #offset]! | EA=[Rn]+offset; Rn←[Rn]+offset |
| Post-indexed | [Rn], #offest | EA=[Rn]; Rn←[Rn]+offset |
| With offset in Rn | | |
| Pre-indexed | [Rn, ±Rm, shift] | EA=[Rn]±[Rm] shifted |
| Pre-indexed with writeback | [Rn, ±Rm, shift]! | EA=[Rn]±[Rm] shifted;<br>Rn←[Rn]±[Rm] shifted |
| Post-indexed | [Rn], ±Rm, shift | EA=[Rn];<br>Rn←[Rn]±[Rm] shifted |
| Relative (Pre-indexed with Immediate offset) | Location | EA=Location=[PC]+offset |

shift=direction #integer, where direction is LSL for left shift or LSR for right shift, and integer is a 5-bit unsigned number specifying the shift format
± Rm=the offset magnitude in register Rm can be added to or subtracted from the contents of based register Rn

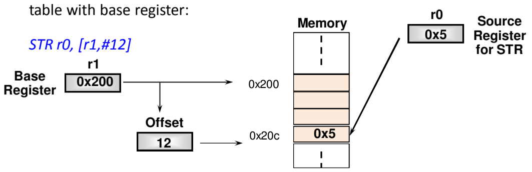
*Advanced Reliable Systems (ARES) Lab.* *Jin-Fu Li, EE, NCU*

# Pre-indexed

*LDR/STR <Rd>,<addressing_mode>*

***Pre-indexed Addressing Mode***

- Mechanism to provide indexed access to a table. Allows easy access to a particular item in the table. For example, to access the forth item in a table with base register:



- The ***offset register*** (R1 in this example) is not changed after instruction execution
- This addressing mode avoids hard-coding. When the table is moved to a different location, we simply need to update the ***base register*** with the new table starting address

## Pre-indexed

*LDR/STR <Rd>,<addressing_mode>*

***Pre-indexed Addressing Mode***

More examples:

| | |
|---|---|
| • LDR R0, [R1, #4] | R0 ← M[R1+4].<br>Registers R1 are not changed |
| • LDR R0, [R1, R2] | R0 ← M[R1+R2]<br>Registers R1 and R2 are not changed |
| • LDR R0, [R1, R2, LSL#2] | R1 ← M[R1 + LSL(R2, 2)]<br>Registers R1 and R2 are not changed |

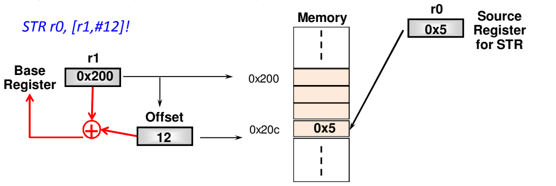
## Pre-indexed with WriteBack

*LDR/STR <Rd>,<addressing_mode>*

***Pre-indexed Addressing Mode with WriteBack***

- Same as Pre-indexed addressing Mode. The difference is that the offset register will be updated **after** the operation is completed.

*STR r0, [r1,#12]!*



- The *offset register* (R1 in this example) is updated after the operation

*R1 ← R1 + offset*

- Useful to implement a loop to perform sequential access to all items in the table

## Pre-indexed with WriteBack

*LDR/STR <Rd>,<addressing_mode>*

***Pre-indexed Addressing Mode with WriteBack***

More examples:

| | |
|---|---|
| • LDR R0, [R1, #4]! | R0 ← M[R1+4]<br>R1 ← R1 + 4 |
| • LDR R0, [R1, R2]! | R0 ← M[R1+ R2]<br>R1 ← R1 + R2<br>R2 is unchanged |
| • LDR R0, [R1, R2, LSL #2]! | R0 ← M[R1+LSL(R2,2)]<br>R1 ←R1 + LSL(R2, 2)<br>R2 is unchanged |

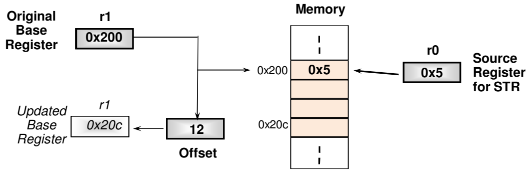
## Post-Indexed

*LDR/STR <Rd>,<addressing_mode>*

***Post-indexed Addressing Mode***

- Different from pre-indexed addressing mode, in post-indexed addressing, ***the effective address is not modified by the offset***.

*STR r0, [r1] , #12*



- The offset register is updated only after the operation

*R1 ← R1 + offset*

## Post-indexed

*LDR/STR <Rd>,<addressing_mode>*

***Post-indexed Addressing Mode***

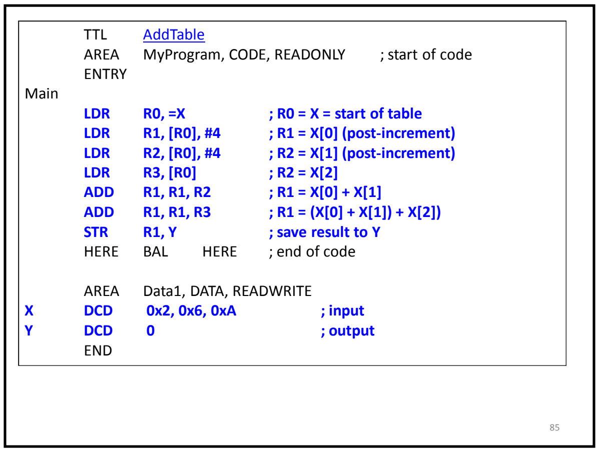
More examples:

| | |
|---|---|
| • LDR R0, [R1], #4 | R0 ← M[R1]<br>R1 ← R1 + 4 |
| • LDR R0, [R1], R2 | R0 ← M[R1]<br>R1 ← R1 + R2<br>R2 is unchanged |
| • LDR R0, [R1], R2, LSL #2 | R0 ← M[R1]<br>R1 ← R1 + LSL(R2,2)<br>R2 is unchanged |

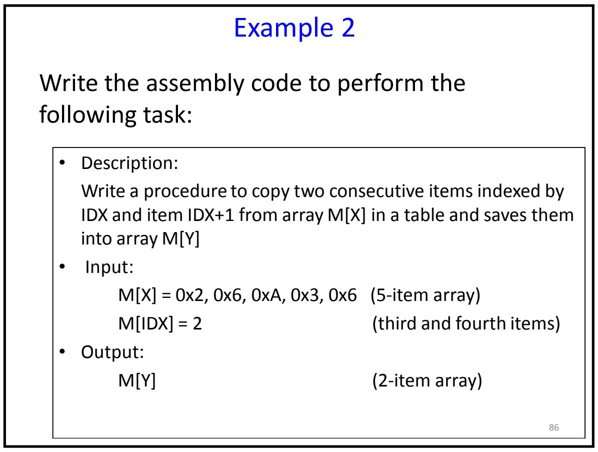
## Example 1

Write the assembly code to perform the following task:

- Description:
  Write a procedure which adds all items in a table of data (word size).
- Input:
  M[X] = 0x2, 0x6, 0xA (table)
- Output:
  M[Y] (sum of items)



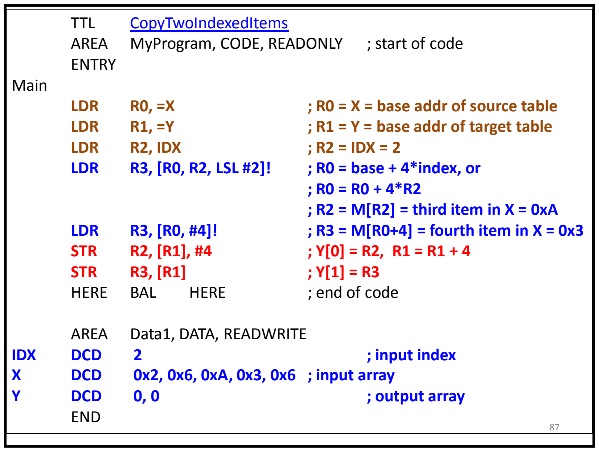
```
        TTL     AddTable
        AREA    MyProgram, CODE, READONLY       ; start of code
        ENTRY
Main
        LDR     R0, =X          ; R0 = X = start of table
        LDR     R1, [R0], #4    ; R1 = X[0] (post-increment)
        LDR     R2, [R0], #4    ; R2 = X[1] (post-increment)
        LDR     R3, [R0]        ; R2 = X[2]
        ADD     R1, R1, R2      ; R1 = X[0] + X[1]
        ADD     R1, R1, R3      ; R1 = (X[0] + X[1]) + X[2])
        STR     R1, Y           ; save result to Y
        HERE    BAL     HERE    ; end of code

        AREA    Data1, DATA, READWRITE
X       DCD      0x2, 0x6, 0xA           ; input
Y       DCD      0                       ; output
        END
```



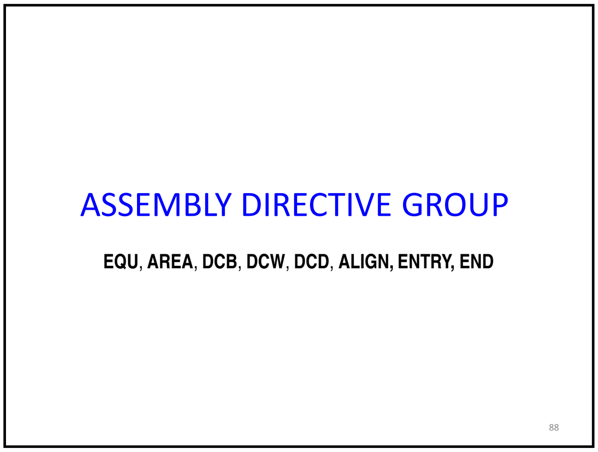
## Example 2

Write the assembly code to perform the following task:

- Description:
  Write a procedure to copy two consecutive items indexed by IDX and item IDX+1 from array M[X] in a table and saves them into array M[Y]
- Input:
  M[X] = 0x2, 0x6, 0xA, 0x3, 0x6 (5-item array)
  M[IDX] = 2 (third and fourth items)
- Output:
  M[Y] (2-item array)

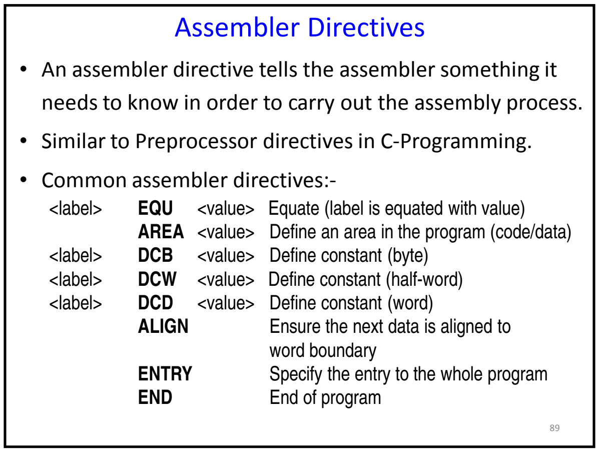


```
        TTL     CopyTwoIndexedItems
        AREA    MyProgram, CODE, READONLY    ; start of code
        ENTRY
Main
        LDR     R0, =X                  ; R0 = X = base addr of source table
        LDR     R1, =Y                  ; R1 = Y = base addr of target table
        LDR     R2, IDX                 ; R2 = IDX = 2
        LDR     R3, [R0, R2, LSL #2]!   ; R0 = base + 4*index, or
                                        ; R0 = R0 + 4*R2
                                        ; R2 = M[R2] = third item in X = 0xA
        LDR     R3, [R0, #4]!           ; R3 = M[R0+4] = fourth item in X = 0x3
        STR     R2, [R1], #4            ; Y[0] = R2,  R1 = R1 + 4
        STR     R3, [R1]                ; Y[1] = R3
        HERE    BAL     HERE            ; end of code

        AREA    Data1, DATA, READWRITE
IDX     DCD      2                              ; input index
X       DCD      0x2, 0x6, 0xA, 0x3, 0x6 ; input array
Y       DCD      0, 0                           ; output array
        END
```



# ASSEMBLY DIRECTIVE GROUP

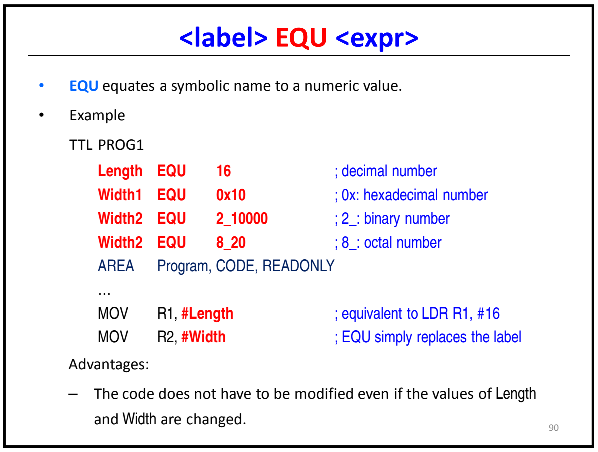
**EQU, AREA, DCB, DCW, DCD, ALIGN, ENTRY, END**



## Assembler Directives

- An assembler directive tells the assembler something it needs to know in order to carry out the assembly process.
- Similar to Preprocessor directives in C-Programming.
- Common assembler directives:-

| | | | |
|---|---|---|---|
| <label> | **EQU** | <value> | Equate (label is equated with value) |
| | **AREA** | <value> | Define an area in the program (code/data) |
| <label> | **DCB** | <value> | Define constant (byte) |
| <label> | **DCW** | <value> | Define constant (half-word) |
| <label> | **DCD** | <value> | Define constant (word) |
| | **ALIGN** | | Ensure the next data is aligned to word boundary |
| | **ENTRY** | | Specify the entry to the whole program |
| | **END** | | End of program |



## <label> EQU <expr>

- EQU equates a symbolic name to a numeric value.
- Example

```
TTL PROG1
      Length   EQU    16              ; decimal number
      Width1   EQU    0x10            ; 0x: hexadecimal number
      Width2   EQU    2_10000         ; 2_: binary number
      Width2   EQU    8_20            ; 8_: octal number
      AREA     Program, CODE, READONLY
      ...
      MOV      R1, #Length            ; equivalent to LDR R1, #16
      MOV      R2, #Width             ; EQU simply replaces the label
```

Advantages:

– The code does not have to be modified even if the values of Length and Width are changed.

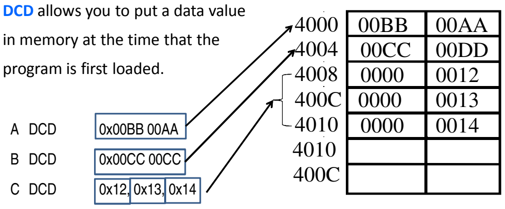
## AREA <name>, <atr> <atr>, ENTRY, END

- AREA establishes indivisible memory regions that are manipulated by the linker. Key attributes include
  - CODE: area includes only instruction
  - DATA: area includes only data
  - READONLY: default for CODE areas
  - READWRITE: default for DATA areas
- ENTRY indicates the point in the code where program execution should begin. There should be only ONE entry point per complete program.
- END tells the assembler that the end of a program is reached.

```
AREA        Prog1, CODE, READONLY       ; define a new code region labeled as Prog1
ENTRY                                   ; Entry point for program
...                                     ; code write your code here

AREA        Table1, DATA , READWRITE    ; define a new data region labeled as Table1
...                                     ; data goes here
END                                     ; End of program
```

## <label> DCD <expr>

- DCD allocates words (32 bit) of memory, padding as necessary to ensure word alignment and initializes them to the values given
- DCD allows you to put a data value in memory at the time that the program is first loaded.



```
A  DCD    0x00BB 00AA
B  DCD    0x00CC 00CC
C  DCD    0x12, 0x13, 0x14
```

| | | |
|---|---|---|
| 4000 | 00BB | 00AA |
| 4004 | 00CC | 00DD |
| 4008 | 0000 | 0012 |
| 400C | 0000 | 0013 |
| 4010 | 0000 | 0014 |
| 4010 | | |
| 400C | | |

A, B and C is actually pointers (point to a memory address). The actual value of A = 0x4000 and B = 0x4004. The content pointed by A or M[A] is 0x00BB 00AA

## <label> DCW/DCB <expr>, ALIGN

- DCW allocates half-words (16 bit) of memory and initializes it with the value
- DCB allocates byte (8 bytes) of memory
- ALIGN e-aligns the data so that the next data will be aligned to the word boundary

```
A   DCW    0xBBBB

B   DCB    0xAA
    ALIGN

C   DCB    "ABC"
    ALIGN

D   DCD    0xCCCCCCCC
```

| | | |
|---|---|---|
| 4000 | BBBB | AA *00* |
| 4004 | 4142 | 43 *00* |
| 4008 | CCCC | CCCC |
| 400C | | |
| 4010 | | |
| 4010 | | |
| 400C | | |

## DATA TRANSFER GROUP

– Register to Register (MOV, MVN)

– Memory to Register (LDR, LDM)

– Register to Memory (STR, STM)

## MOV<cond><s> <Rd>,<Shifter Operand>

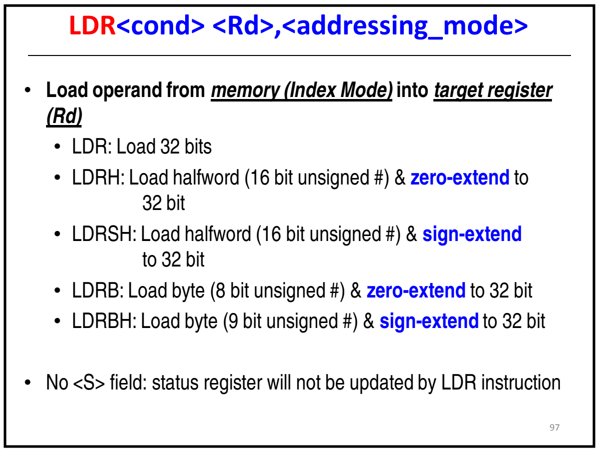
- Performs a move to a register (Rd) the content from another register (with barrel shift) or an immediate value (32 bit)
- Flags updated if S used and Rd is not R15: N, Z and C
- Example:

```
MOV R1, R0, LSL #2          ;R1 <- R0*4
Status flags not updated

MOVS R1, #0                 ;R1<- 0x00000001
Status flags – N:0, Z:1, V:0, C:0

MOVSNE  R0, #0x4
if status flag N = 0, the R0 ← 0x00000004
If status flag N = 1, instruction is ignored
```

## MVN<cond><s> <Rd>,<Shifter Operand>

- MVN complements the value of a register or an immediate value and stores it in the destination register (Rd)
- Flags updated if S used and Rd is not R15: N, Z and C
- Example:

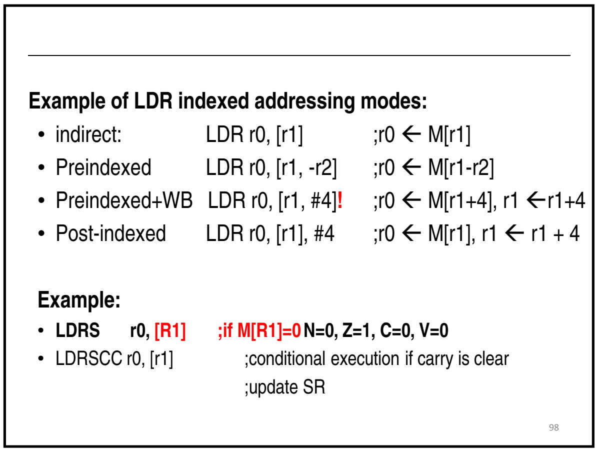
```
MVN R1, R0, LSL #2          ;R1 <- NOT(R0*4)
Status flag is not updated

MVNS R1, #1                 ;R1<- FFFFFFFE
Status flags – N:1, Z:0, V:0, C:0
```



## LDR<cond> <Rd>,<addressing_mode>

- **Load operand from *<u>memory (Index Mode)</u>* into *<u>target register (Rd)</u>***
  - LDR: Load 32 bits
  - LDRH: Load halfword (16 bit unsigned #) & **zero-extend** to 32 bit
  - LDRSH: Load halfword (16 bit unsigned #) & **sign-extend** to 32 bit
  - LDRB: Load byte (8 bit unsigned #) & **zero-extend** to 32 bit
  - LDRBH: Load byte (9 bit unsigned #) & **sign-extend** to 32 bit
- No <S> field: status register will not be updated by LDR instruction

97



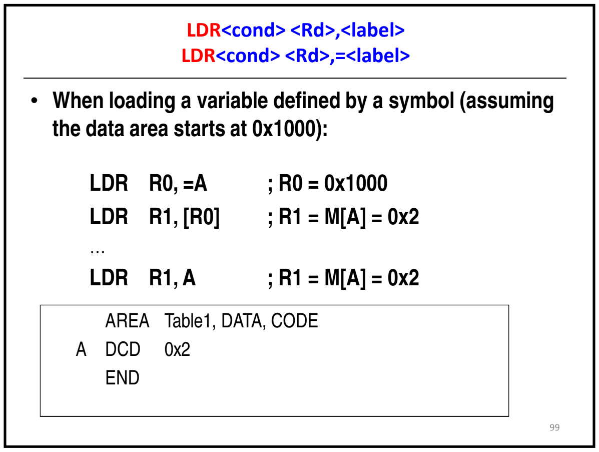
**Example of LDR indexed addressing modes:**

```
• indirect:        LDR r0, [r1]          ;r0 ← M[r1]
• Preindexed       LDR r0, [r1, -r2]     ;r0 ← M[r1-r2]
• Preindexed+WB    LDR r0, [r1, #4]!     ;r0 ← M[r1+4], r1 ←r1+4
• Post-indexed     LDR r0, [r1], #4      ;r0 ← M[r1], r1 ← r1 + 4
```

**Example:**

```
• LDRS     r0, [R1]      ;if M[R1]=0 N=0, Z=1, C=0, V=0
• LDRSCC r0, [r1]            ;conditional execution if carry is clear
                             ;update SR
```

98



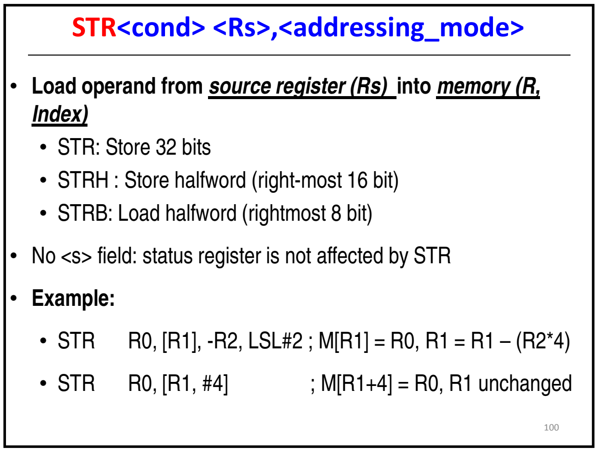
## LDR<cond> <Rd>,<label>
## LDR<cond> <Rd>,=<label>

- **When loading a variable defined by a symbol (assuming the data area starts at 0x1000):**

```
LDR   R0, =A        ; R0 = 0x1000
LDR   R1, [R0]      ; R1 = M[A] = 0x2
…
LDR   R1, A         ; R1 = M[A] = 0x2
```

```
      AREA   Table1, DATA, CODE
A     DCD    0x2
      END
```

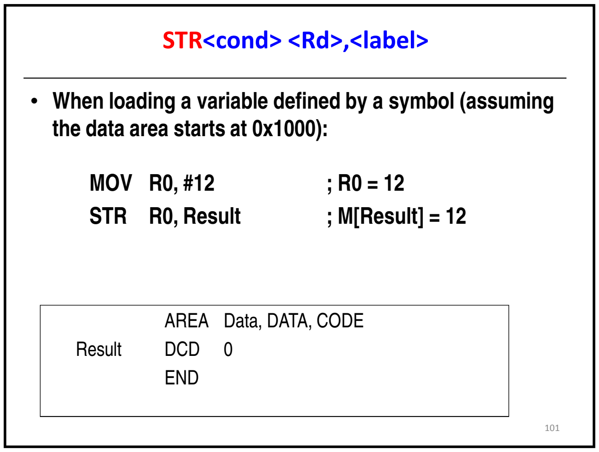
99



## STR<cond> <Rs>,<addressing_mode>

- **Load operand from *<u>source register (Rs)</u>* into *<u>memory (R, Index)</u>***
  - STR: Store 32 bits
  - STRH : Store halfword (right-most 16 bit)
  - STRB: Load halfword (rightmost 8 bit)
- No <s> field: status register is not affected by STR
- **Example:**
  - STR R0, [R1], -R2, LSL#2 ; M[R1] = R0, R1 = R1 – (R2*4)
  - STR R0, [R1, #4] ; M[R1+4] = R0, R1 unchanged

100



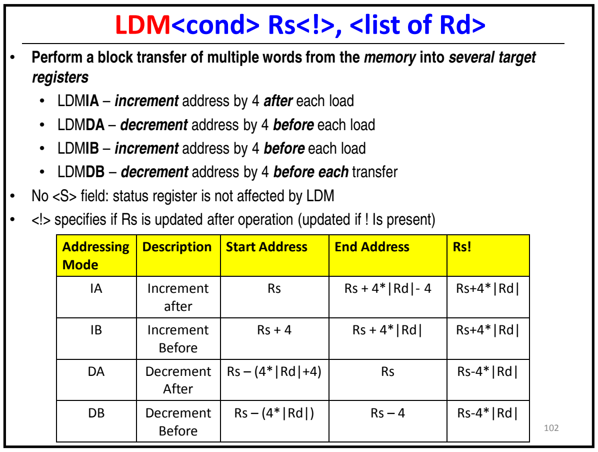
## STR<cond> <Rd>,<label>

- **When loading a variable defined by a symbol (assuming the data area starts at 0x1000):**

```
MOV   R0, #12          ; R0 = 12
STR   R0, Result       ; M[Result] = 12
```

```
          AREA   Data, DATA, CODE
Result    DCD    0
          END
```

101

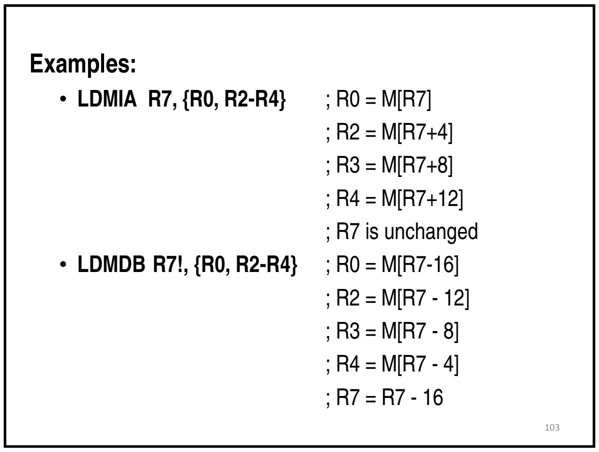


## LDM<cond> Rs<!>, <list of Rd>

- **Perform a block transfer of multiple words from the *memory* into *several target registers***
  - LDM**IA** – ***increment*** address by 4 ***after*** each load
  - LDM**DA** – ***decrement*** address by 4 ***before*** each load
  - LDM**IB** – ***increment*** address by 4 ***before*** each load
  - LDM**DB** – ***decrement*** address by 4 ***before each*** transfer
- No <S> field: status register is not affected by LDM
- <!> specifies if Rs is updated after operation (updated if ! Is present)

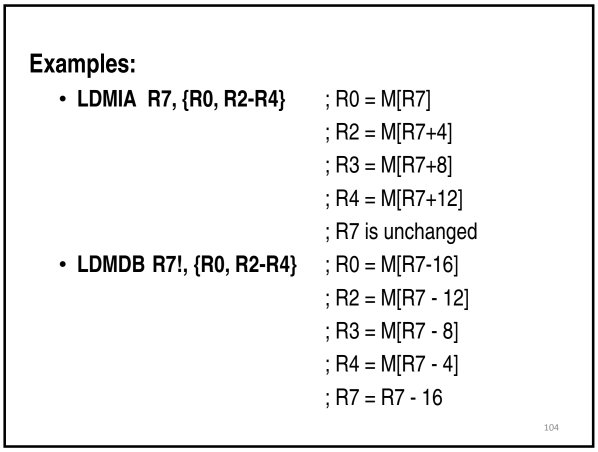
| Addressing Mode | Description | Start Address | End Address | Rs! |
|---|---|---|---|---|
| IA | Increment after | Rs | Rs + 4*\|Rd\|- 4 | Rs+4*\|Rd\| |
| IB | Increment Before | Rs + 4 | Rs + 4*\|Rd\| | Rs+4*\|Rd\| |
| DA | Decrement After | Rs – (4*\|Rd\|+4) | Rs | Rs-4*\|Rd\| |
| DB | Decrement Before | Rs – (4*\|Rd\|) | Rs – 4 | Rs-4*\|Rd\| |

102



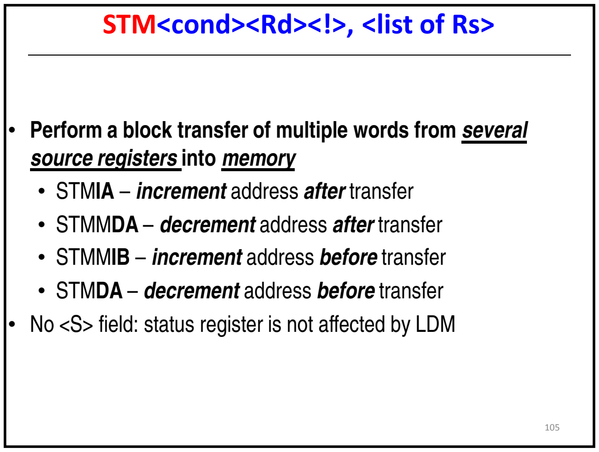
**Examples:**

```
• LDMIA  R7, {R0, R2-R4}      ; R0 = M[R7]
                              ; R2 = M[R7+4]
                              ; R3 = M[R7+8]
                              ; R4 = M[R7+12]
                              ; R7 is unchanged
• LDMDB R7!, {R0, R2-R4}      ; R0 = M[R7-16]
                              ; R2 = M[R7 - 12]
                              ; R3 = M[R7 - 8]
                              ; R4 = M[R7 - 4]
                              ; R7 = R7 - 16
```



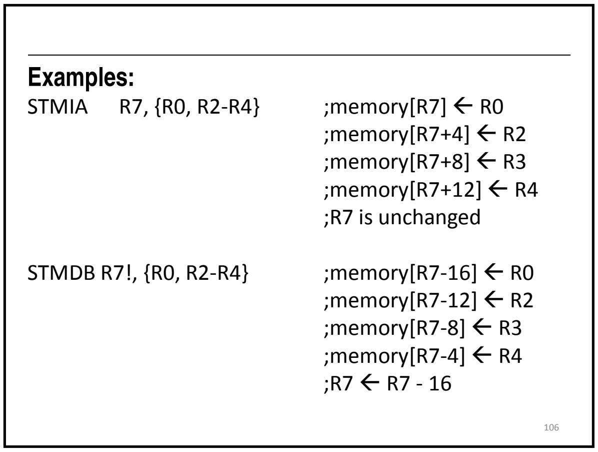
**Examples:**

```
• LDMIA  R7, {R0, R2-R4}      ; R0 = M[R7]
                              ; R2 = M[R7+4]
                              ; R3 = M[R7+8]
                              ; R4 = M[R7+12]
                              ; R7 is unchanged
• LDMDB R7!, {R0, R2-R4}      ; R0 = M[R7-16]
                              ; R2 = M[R7 - 12]
                              ; R3 = M[R7 - 8]
                              ; R4 = M[R7 - 4]
                              ; R7 = R7 - 16
```



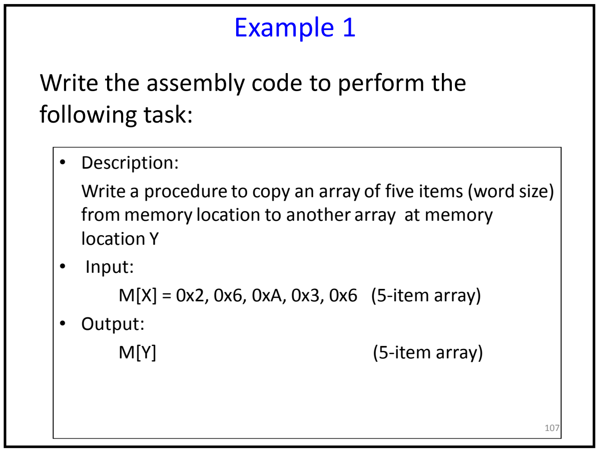
## STM<cond><Rd><!>, <list of Rs>

- **Perform a block transfer of multiple words from *__several source registers__* into *__memory__***
  - **STMIA** – ***increment*** address ***after*** transfer
  - STMM**DA** – ***decrement*** address ***after*** transfer
  - STMM**IB** – ***increment*** address ***before*** transfer
  - STM**DA** – ***decrement*** address ***before*** transfer
- No <S> field: status register is not affected by LDM



**Examples:**

```
STMIA     R7, {R0, R2-R4}        ;memory[R7] ← R0
                                 ;memory[R7+4] ← R2
                                 ;memory[R7+8] ← R3
                                 ;memory[R7+12] ← R4
                                 ;R7 is unchanged

STMDB R7!, {R0, R2-R4}           ;memory[R7-16] ← R0
                                 ;memory[R7-12] ← R2
                                 ;memory[R7-8] ← R3
                                 ;memory[R7-4] ← R4
                                 ;R7 ← R7 - 16
```



## Example 1

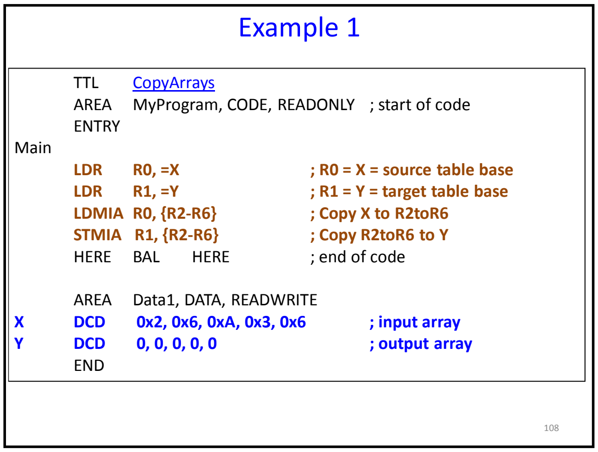
Write the assembly code to perform the following task:

- Description:

  Write a procedure to copy an array of five items (word size) from memory location to another array at memory location Y
- Input:

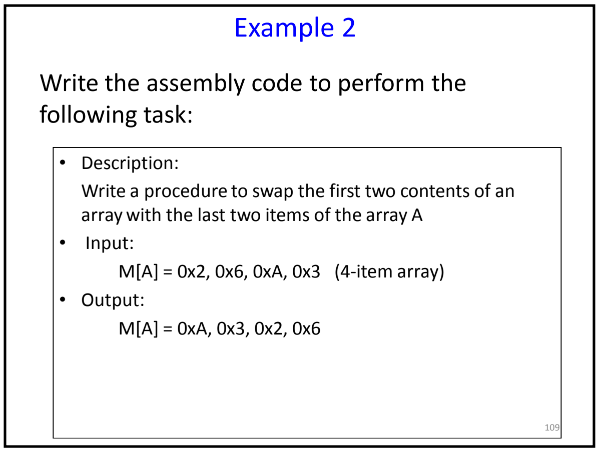
  M[X] = 0x2, 0x6, 0xA, 0x3, 0x6 (5-item array)
- Output:

  M[Y] (5-item array)



## Example 1

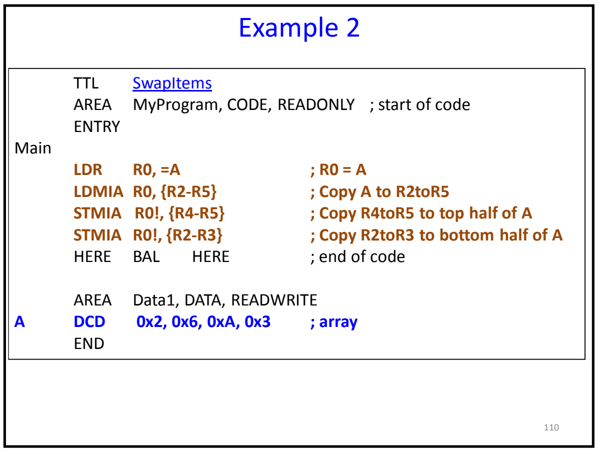
```
        TTL     CopyArrays
        AREA    MyProgram, CODE, READONLY   ; start of code
        ENTRY
Main
        LDR     R0, =X                 ; R0 = X = source table base
        LDR     R1, =Y                 ; R1 = Y = target table base
        LDMIA   R0, {R2-R6}            ; Copy X to R2toR6
        STMIA   R1, {R2-R6}            ; Copy R2toR6 to Y
        HERE    BAL     HERE           ; end of code

        AREA    Data1, DATA, READWRITE
X       DCD     0x2, 0x6, 0xA, 0x3, 0x6        ; input array
Y       DCD     0, 0, 0, 0, 0                  ; output array
        END
```



## Example 2

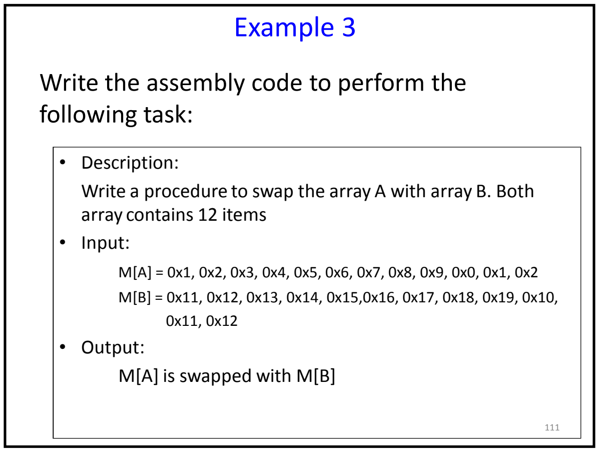
Write the assembly code to perform the following task:

- Description:
  Write a procedure to swap the first two contents of an array with the last two items of the array A
- Input:
  M[A] = 0x2, 0x6, 0xA, 0x3 (4-item array)
- Output:
  M[A] = 0xA, 0x3, 0x2, 0x6



## Example 2

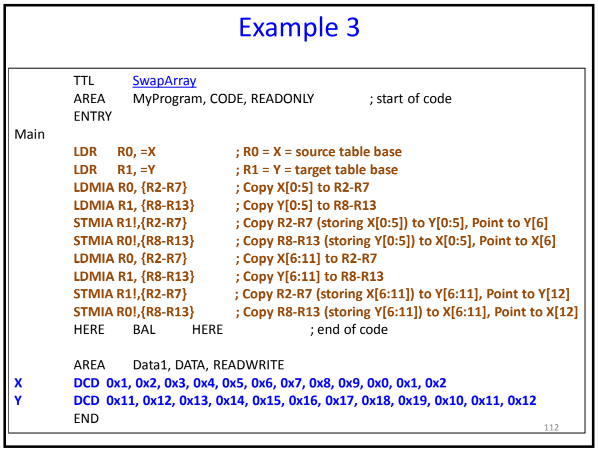
```
        TTL     SwapItems
        AREA    MyProgram, CODE, READONLY   ; start of code
        ENTRY
Main
        LDR     R0, =A                  ; R0 = A
        LDMIA   R0, {R2-R5}             ; Copy A to R2toR5
        STMIA   R0!, {R4-R5}            ; Copy R4toR5 to top half of A
        STMIA   R0!, {R2-R3}            ; Copy R2toR3 to bottom half of A
        HERE    BAL     HERE            ; end of code

        AREA    Data1, DATA, READWRITE
A       DCD     0x2, 0x6, 0xA, 0x3      ; array
        END
```



## Example 3

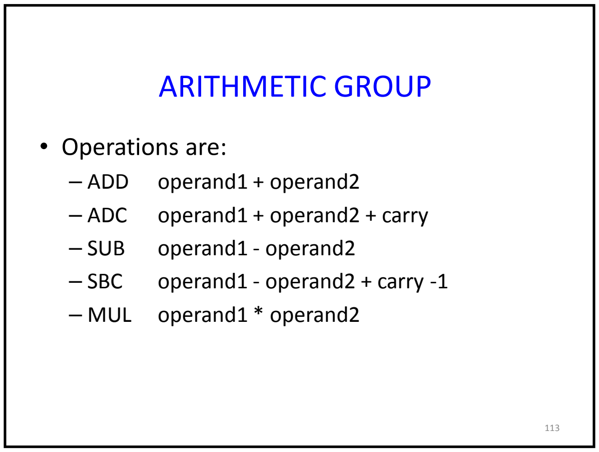
Write the assembly code to perform the following task:

- Description:
  Write a procedure to swap the array A with array B. Both array contains 12 items
- Input:
  M[A] = 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0x0, 0x1, 0x2
  M[B] = 0x11, 0x12, 0x13, 0x14, 0x15,0x16, 0x17, 0x18, 0x19, 0x10, 0x11, 0x12
- Output:
  M[A] is swapped with M[B]



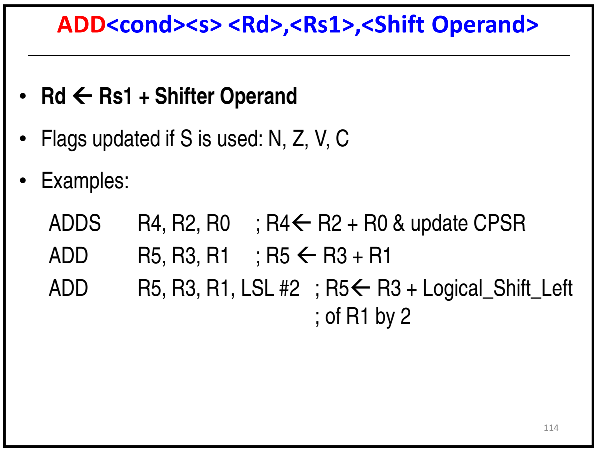
## Example 3

```
        TTL     SwapArray
        AREA    MyProgram, CODE, READONLY       ; start of code
        ENTRY
Main
        LDR     R0, =X           ; R0 = X = source table base
        LDR     R1, =Y           ; R1 = Y = target table base
        LDMIA R0, {R2-R7}        ; Copy X[0:5] to R2-R7
        LDMIA R1, {R8-R13}       ; Copy Y[0:5] to R8-R13
        STMIA R1!,{R2-R7}        ; Copy R2-R7 (storing X[0:5]) to Y[0:5], Point to Y[6]
        STMIA R0!,{R8-R13}       ; Copy R8-R13 (storing Y[0:5]) to X[0:5], Point to X[6]
        LDMIA R0, {R2-R7}        ; Copy X[6:11] to R2-R7
        LDMIA R1, {R8-R13}       ; Copy Y[6:11] to R8-R13
        STMIA R1!,{R2-R7}        ; Copy R2-R7 (storing X[6:11]) to Y[6:11], Point to Y[12]
        STMIA R0!,{R8-R13}       ; Copy R8-R13 (storing Y[6:11]) to X[6:11], Point to X[12]
        HERE    BAL     HERE             ; end of code

        AREA    Data1, DATA, READWRITE
X       DCD  0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0x0, 0x1, 0x2
Y       DCD  0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x10, 0x11, 0x12
        END
```



## ARITHMETIC GROUP

- Operations are:
  - ADD operand1 + operand2
  - ADC operand1 + operand2 + carry
  - SUB operand1 - operand2
  - SBC operand1 - operand2 + carry -1
  - MUL operand1 * operand2



## ADD<cond><s> <Rd>,<Rs1>,<Shift Operand>

- **Rd ← Rs1 + Shifter Operand**
- Flags updated if S is used: N, Z, V, C
- Examples:

```
ADDS    R4, R2, R0          ; R4← R2 + R0 & update CPSR
ADD     R5, R3, R1          ; R5 ← R3 + R1
ADD     R5, R3, R1, LSL #2  ; R5← R3 + Logical_Shift_Left
                            ; of R1 by 2
```

## ADC<cond><s> <Rd>,<Rs1>,<Shift Operand>

- Addition with carry: **Rd ← Rs1 + Shifter Operand + C**
- The value of C is determined by previous instructions
- Flags updated if S is used: N, Z, V, C
- The ADC instruction is used to implement efficient multiword addition. For example, if 64-bit numbers are stored in R1:R0 and R3:R2, their sum can be stored in R5:R4 as shown below.

```
ADDS  R4, R2, R0        ; add the least significant word
                        ; field <S> will update the carry bit
ADC   R5, R3, R1        ; add the most significant word
```

## SUB<cond><s> <Rd>,<Rs1>,<Shift Operand>

- **Rd ← Rs1 - Shifter Operand**
- Flags updated if S is used: N, Z, V, C
- Examples:

```
SUBS R0, R0, #1              ; decrement R0 by 1, update flags
SUB   R0, R2, R1             ; R0 ← R2 – R1

SUB   R0, R2, R1, LSR #2     ;  R0 ← R2 – Logical_shift_right
                             ; of  R1 by 2
```

## SBC<cond><s> <Rd>,<Rs1>,<Shift Operand>

- Subtract with carry: **Rd ← Rs1 - Shifter Operand – NOT C**
- ***The value of C is determined by previous instructions.***
- ***For subtraction, ARM clears the carry if the result is less than 0 (borrow required). The carry flag (C) is the inverse of a borrow flag. Therefore, if a borrow is required by the operation, C will be set to 0 and SBC will subtract an additional one from current value.***
- Flags updated if S is used: N, Z, V, C
- The SUB instruction is used to implement efficient multiword subtraction. For example, if 64-bit numbers are stored in R1:R0 and R3:R2, their difference can be stored in R5:R4 as shown below.

```
SUBS R4, R2, R0        ; subtract least significant words
SBC R5, R3, R1         ; subtract most significant words
                         minus borrow
```

## MUL<cond><s> <Rd>,<Rs1>,<Shift Operand>

- **Rd ← Rs1 * Shift_Operand**
- Flags updated if S is used: N, Z
- MUL performs a 32x32 operation, and stores the lower 32 bit of the result into Rd.
- Since only the least significant 32-bits are stored, the result may not be the complete → Suitable only for half-word. To overcome this instruction, use the UMULL instruction.
- Example:

```
MUL R0, R1, R2      ;R0 ← R1*R2
```

## LOGICAL OPERATION

– AND operand1 AND operand2
– EOR operand1 EOR operand2
– ORR operand1 OR operand2
– BIC operand1 AND NOT operand2

## AND, EOR, ORR, BIC <cond><s> <Rd>,<Rs1>,<Shift Operand>

- Syntax:
  <Operation>{<cond>}{S} Rd, Rn, Shift_Operand
- Flags updated if S used: N, Z, C
- Logical operations can be used to manipulate the data. Examples:
  - **AND** R0, R0, #0x8000 ; **mask** bit D15 of R0
  - **EOR** R0, R0, #0x8000 ; **toggle** bit D15 of R0
  - **ORR** R0, R0, #0x8000 ; **set** bit D15 of R0
  - **BIC** R0, R0, #0x8000 ; **clears** bit D15of R0

## Example 1

Write the assembly code to perform the following task:

- Description:

  Write a procedure to add three numbers in memory location *Table*. Store the result in memory location *Result*.
- Input:

  M[Table] = 0x1, 0x3, 0x4 (3-item array)
- Output:

  M[Result]

```
        TTL     Add3No
        AREA    MyProgram, CODE, READONLY    ; start of code
        ENTRY
Main
        LDR     R0, =Table              ; Initialize R0 to start of table
        LDMIA   R0, {R1-R3}             ; Get three numbers
        ADD     R4, R1, R2              ; Add the first and second number
        ADD     R4, R4, R3              ; Add the third number
        STR     R4, Result              ; Store result
        HERE    BAL     HERE            ; end of code

        AREA    Data1, DATA, READWRITE
Table   DCD     0x1, 0x3, 0x4           ; array
Result  DCD     0                       ; result
        END
```

## Example 2

Write the assembly code to perform the following task:

- Description:

  Write a procedure to add two long words in memory location *A* and *B, respectively*. Store the result in memory location Result.
- Input:

  M[A] = 0x20002000F000F000 (3-item array)

  M[B] = 0x3000300010001000
- Output:

  M[Result]

```
        TTL     AddLongNumbers
        AREA    MyProgram, CODE, READONLY       ; start of code
        ENTRY
Main
        LDR     R0, =A          ; Initialize R0 to MSB of A
        LDMIA   R0, {R1-R2}     ; MSW → R1, LSW →R2
        LDR     R0, =B          ; Initialize R0 to MSB of B
        LDMIA   R0, {R3-R4}     ; MSW → R3, LSW → R4
        ADDS    R6, R2, R4      ; Add the LSW (Update CPSR)
        ADC     R5, R1, R3      ; Add the MSB and carry from LSW
        LDR     R0, =Result     ; Initialize R0 to MSB of Result
        STMIA   R0, {R5-R6}     ; Store long word result to Result
        HERE    BAL     HERE    ; end of code

        AREA    Data1, DATA, READWRITE
A       DCD     0x20002000, 0xF000F000
B       DCD     0x30003000, 0x10001000
Result  DCD     0, 0                    ; result
        END
```

## Example 3

Write the assembly code to perform the following task:

- Description:

  Given a five bit binary code stored in the lowest four bits of M[*A*], convert the binary code into a gray code
- Input:

  M[A] = 0x0000000B (Binary Code)
- Output:

  M[B] = 0x0000000E (Gray Code)

```
        TTL     ConvertToGrayCode
        AREA    MyProgram, CODE, READONLY               ; start of code
        ENTRY
Main
        LDR     R0, A                   ; Initialize R0  = M[A] = 1011_2
        AND     R2, R0, #2_1000         ; R2 = get bit 3         → R2 stores bit 3
        AND     R3, R0, #2_0100         ; R3 = get bit 2
        AND     R4, R0, #2_0010         ; R4 = get bit 1
        AND     R5, R0, #2_0001         ; R5 = get bit 0
        EOR     R6, R3, R2, LSR #1      ; R6 = bit 2 XOR bit 3 →R6 stores bit 2
        EOR     R7, R4, R3, LSR #1      ; R7 = bit 1 XOR bit 2 → R7 stores bit 1
        EOR     R8, R5, R4, LSR #1      ; R8 = bit 0 XOR bit 1 → R8 stores bit 0
        ORR     R2, R2, R6              ; Accumulate all bits
        ORR     R2, R2, R7
        ORR     R2, R2, R8
        STR     R2, B
HERE    BAL     HERE    ; end of code

        AREA    Data1, DATA, READWRITE
A       DCD     0x0000000B
B       DCD     0
        END
```

## COMPARISON GROUP

CMP TST TEQ

## Comparison Group

- The only effect of the comparisons is to
  - ***UPDATE THE CONDITION FLAGS*.** Thus no need to set S bit.
- Operations are:
  - CMP operand1 - operand2, but result not written
  - TST operand1 AND operand2, but result not written
  - TEQ operand1 EOR operand2, but result not written
- Syntax:
  - <Operation>{<cond>} Rn, Operand2
- Examples:
  - CMP r0, r1
  - TSTEQ r2, #5

## CMP<cond> <Rn>,<shifter operand>

- The CMP instruction performs a subtraction, but does not store the result. The flags are always updated.
- **Rn – Shifter_operand**
- Examples:

```
CMP R0, #1      ; Z=1 if R0 = 1, Z = 0 if R0 > 1

CMP R0, R1      ; Z =1  if R0 = R1, Z = 0 if R0 > R1
```

## TST <cond> <Rn>,<shifter_operand>

- The TST instruction performs a **non-destructive AND** (the result is not stored). The flags are always updated.
- The most common use for this instruction is to determine the value of an individual bit of a register.
- **Rn AND shifter_operand**
- Examples:

```
TST R0, #0x8000         ;if bit 15 of R0 = 0, set Z = 1
```

## TEQ <cond><Rn>,<shifter_operand>

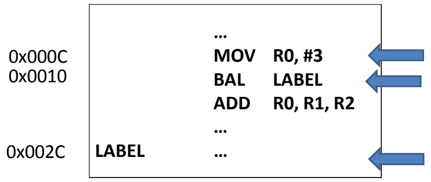
- The TEQ instruction performs a **non-destructive bit-wise XOR** (the result is not stored). The flags are always updated.
- The most common use for this instruction is to determine if two operands are equal without affecting the V flag. If equal, Z = 1
- It can also be use to tell if two values have the same sign, since the N flag will be the logical XOR of the two sign bits.
- **Rn XOR shifter_operand**
- Examples:

```
TEQ R0, #0x8000    ; Z = 1 if R0 = 0x00008000
TEQ R0, R1         ; N = 1 if the signs are different
```

## FLOW CONTROL GROUP

B, BL (not covered)

## Flow Control Group



```
                       ...
0x000C                 MOV    R0, #3
0x0010                 BAL    LABEL
                       ADD    R0, R1, R2
                       ...
0x002C        LABEL    ...
```

- In general, an instruction is executed in sequence until it bumps into a branch instruction.
- If the instruction in 0x0010 is not BAL, the sequence of value for Program Counter (PC) is 0x0010 → (0x0010 + 4 = 0x0014) → 0x0018
- With the BAL (always branch) instruction, the sequence of value for PC becomes 0x0010 → 0x002C where 0x002C is the address of the target destination label

## Flow Control Group

B:

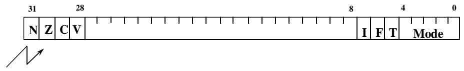
- Branch to the location specified by the label
- Does not affect the Link Register (LR)
- Used to implement loop or if-else statements

BL (not covered):

- Used to implement functions (to remember the caller function and location, so that it can return to the right place after running the called function)
- Similar to B but additionally save the value PC-4 into LR of the current bank: *BL LOCATION_LABEL*
- To return from subroutine, simply restore the PC from the Link Register: *MOV PC, LR*

## B<cond> <target address>

- A branch instruction tests a branch condition and then, depending on the result, causes execution to proceed along one of two possible paths.
- The branch conditions are related to the result of a recently performed operation which updates the status register



* **Condition Code Flags**

N = **N**egative result from ALU flag.
Z = **Z**ero result from ALU flag.
C = ALU operation **C**arried out
V = ALU operation o**V**erflowed

## B<cond> <target address>

The B<cond> instructions are used to branch to a target address, based on an optional condition <cond>. Possible values for <cond> are as follows:

| Opcode [31:28] | Mnemonic extension | Interpretation | Status flag state for execution |
|---|---|---|---|
| 0000 | EQ | Equal / equals zero | Z set |
| 0001 | NE | Not equal | Z clear |
| 0010 | CS/HS | Carry set / unsigned higher or same | C set |
| 0011 | CC/LO | Carry clear / unsigned lower | C clear |
| 0100 | MI | Minus / negative | N set |
| 0101 | PL | Plus / positive or zero | N clear |
| 0110 | VS | Overflow | V set |
| 0111 | VC | No overflow | V clear |
| 1000 | HI | Unsigned higher | C set and Z clear |
| 1001 | LS | Unsigned lower or same | C clear or Z set |
| 1010 | GE | Signed greater than or equal | N equals V |
| 1011 | LT | Signed less than | N is not equal to V |
| 1100 | GT | Signed greater than | Z clear and N equals V |
| 1101 | LE | Signed less than or equal | Z set or N is not equal to V |
| 1110 | AL | Always | any |
| 1111 | NV | Never (do not use!) | none |

## Accessing items in an table or array

- The branch instruction is very useful for a lot of cases. One such cases processing repeated items such as an array
- For example, let's say we want to add numbers in a table:
- The most naïve way is to write the following code :

| Table |
|---|
| 3 |
| 2 |
| 13 |
| 4 |

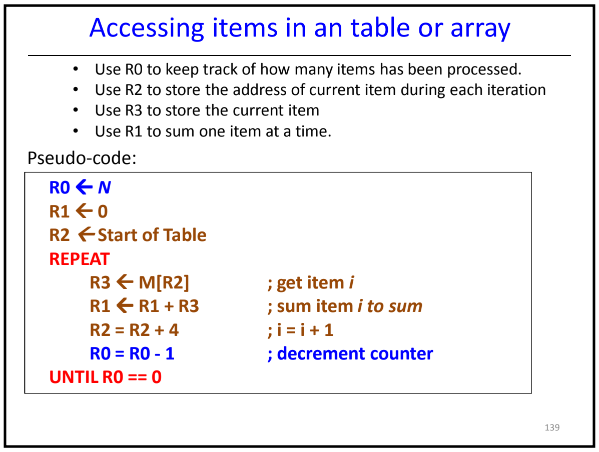
```
LDR  R1, =Table
ADD R0, [R1], #4
ADD R0, [R1], #4
ADD R0, [R1], #4
ADD R0, [R1], #4
…
```

But what if we have 1000 data? (Not a practical solution)

## Accessing items in an table or array

- Solution: use a ***loop***.
  - Put the repetitive code (load and addition) into a loop. Each iteration adds one number.
  - Use a variable to store the total summation value
  - Use a variable to store the unprocessed item in table
  - Use a counter to keep track of how many items has been processed. Exit loop once all items has been processed

```
initialize sum = 0, i = N (#item),
        repeat
                sum = sum + item (N – i + 1)    ; add item i to sum
                i = i - 1                       ; point to next item
                if i == 0, exit from loop       ; all item done?
        end
```

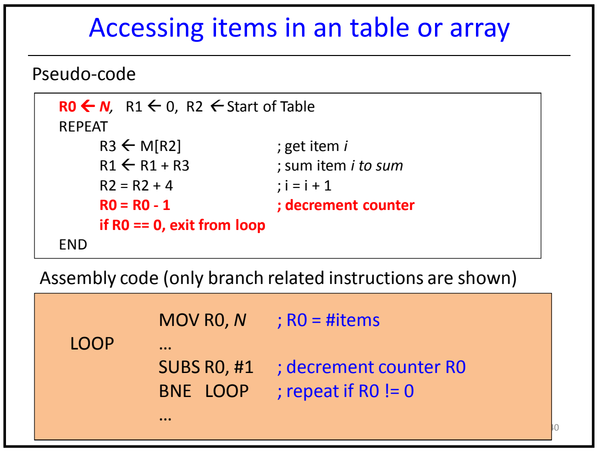


## Accessing items in an table or array

- Use R0 to keep track of how many items has been processed.
- Use R2 to store the address of current item during each iteration
- Use R3 to store the current item
- Use R1 to sum one item at a time.

Pseudo-code:

```
R0 ← N
R1 ← 0
R2 ← Start of Table
REPEAT
      R3 ← M[R2]           ; get item i
      R1 ← R1 + R3         ; sum item i to sum
      R2 = R2 + 4          ; i = i + 1
      R0 = R0 - 1          ; decrement counter
UNTIL R0 == 0
```



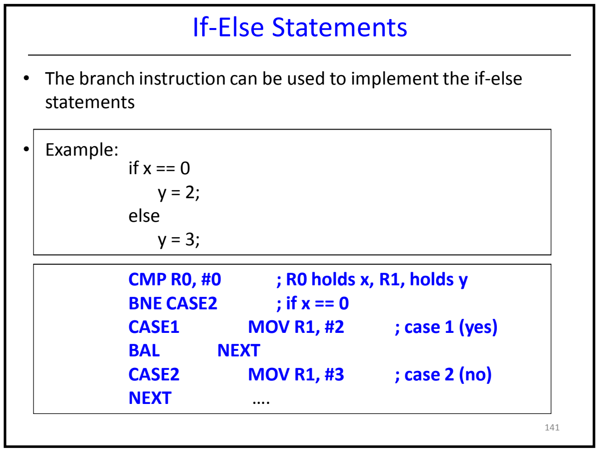
## Accessing items in an table or array

Pseudo-code

```
R0 ← N,   R1 ← 0,  R2 ←Start of Table
REPEAT
      R3 ← M[R2]                 ; get item i
      R1 ← R1 + R3               ; sum item i to sum
      R2 = R2 + 4                ; i = i + 1
      R0 = R0 - 1                ; decrement counter
      if R0 == 0, exit from loop
END
```

Assembly code (only branch related instructions are shown)

```
             MOV R0, N        ; R0 = #items
LOOP         …
             SUBS R0, #1      ; decrement counter R0
             BNE   LOOP       ; repeat if R0 != 0
             …
```



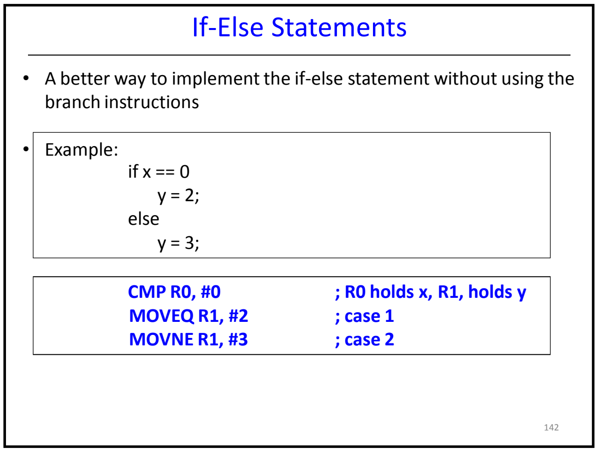
## If-Else Statements

- The branch instruction can be used to implement the if-else statements

- Example:

```
            if x == 0
                 y = 2;
            else
                 y = 3;
```

```
            CMP R0, #0          ; R0 holds x, R1, holds y
            BNE CASE2           ; if x == 0
            CASE1           MOV R1, #2          ; case 1 (yes)
            BAL         NEXT
            CASE2           MOV R1, #3          ; case 2 (no)
            NEXT                ….
```



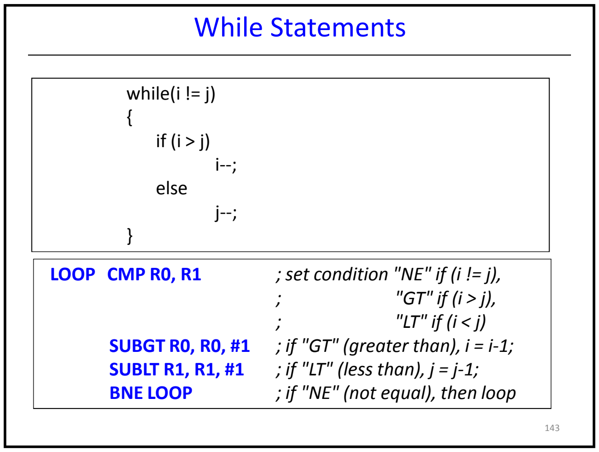
## If-Else Statements

- A better way to implement the if-else statement without using the branch instructions

- Example:

```
            if x == 0
                 y = 2;
            else
                 y = 3;
```

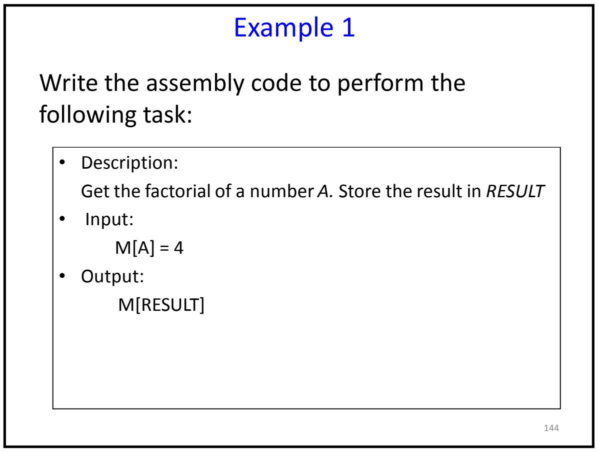
```
            CMP R0, #0                ; R0 holds x, R1, holds y
            MOVEQ R1, #2              ; case 1
            MOVNE R1, #3              ; case 2
```



## While Statements

```
            while(i != j)
            {
                 if (i > j)
                         i--;
                 else
                         j--;
            }
```

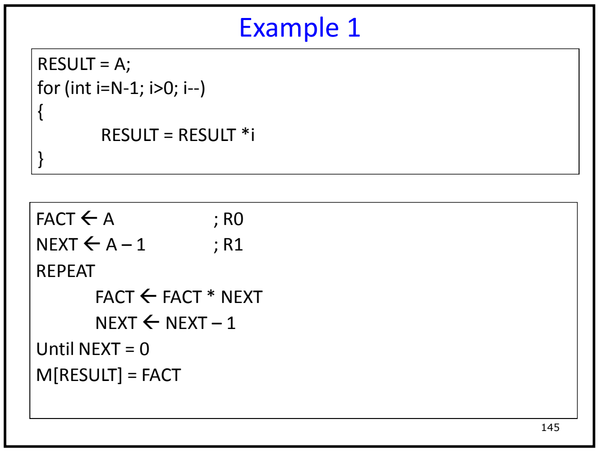
```
LOOP   CMP R0, R1             ; set condition "NE" if (i != j),
                              ;                    "GT" if (i > j),
                              ;                    "LT" if (i < j)
       SUBGT R0, R0, #1       ; if "GT" (greater than), i = i-1;
       SUBLT R1, R1, #1       ; if "LT" (less than), j = j-1;
       BNE LOOP               ; if "NE" (not equal), then loop
```



## Example 1

Write the assembly code to perform the following task:

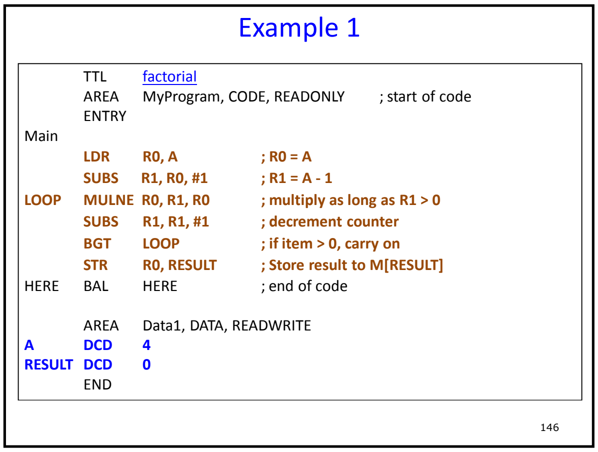
- Description:
  Get the factorial of a number *A*. Store the result in *RESULT*
- Input:
  M[A] = 4
- Output:
  M[RESULT]



## Example 1

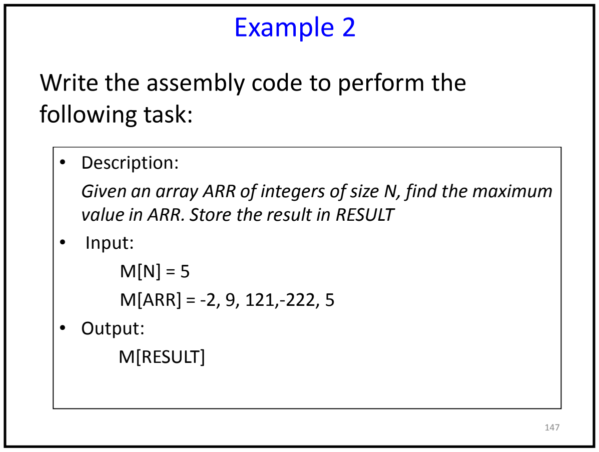
```
RESULT = A;
for (int i=N-1; i>0; i--)
{
        RESULT = RESULT *i
}
```

```
FACT ← A                ; R0
NEXT ← A – 1            ; R1
REPEAT
        FACT ← FACT * NEXT
        NEXT ← NEXT – 1
Until NEXT = 0
M[RESULT] = FACT
```



## Example 1

```
        TTL     factorial
        AREA    MyProgram, CODE, READONLY       ; start of code
        ENTRY
Main
        LDR     R0, A                   ; R0 = A
        SUBS    R1, R0, #1              ; R1 = A - 1
LOOP    MULNE   R0, R1, R0              ; multiply as long as R1 > 0
        SUBS    R1, R1, #1              ; decrement counter
        BGT     LOOP                    ; if item > 0, carry on
        STR     R0, RESULT              ; Store result to M[RESULT]
HERE    BAL     HERE                    ; end of code

        AREA    Data1, DATA, READWRITE
A       DCD     4
RESULT  DCD     0
        END
```



## Example 2

Write the assembly code to perform the following task:

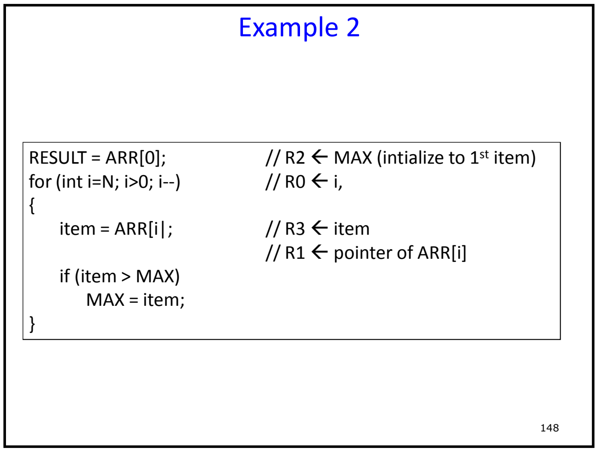
- Description:

  *Given an array ARR of integers of size N, find the maximum value in ARR. Store the result in RESULT*
- Input:

  M[N] = 5

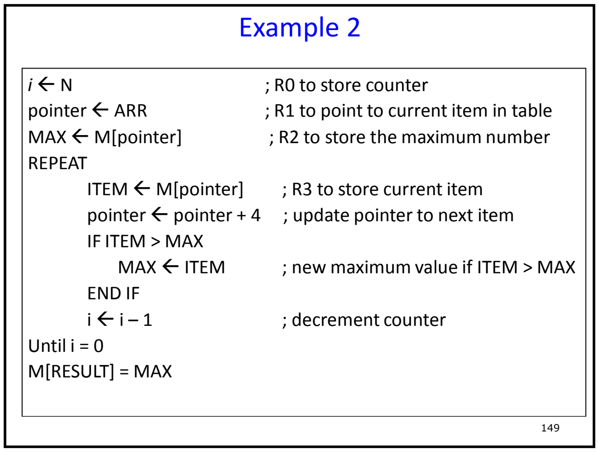
  M[ARR] = -2, 9, 121,-222, 5
- Output:

  M[RESULT]



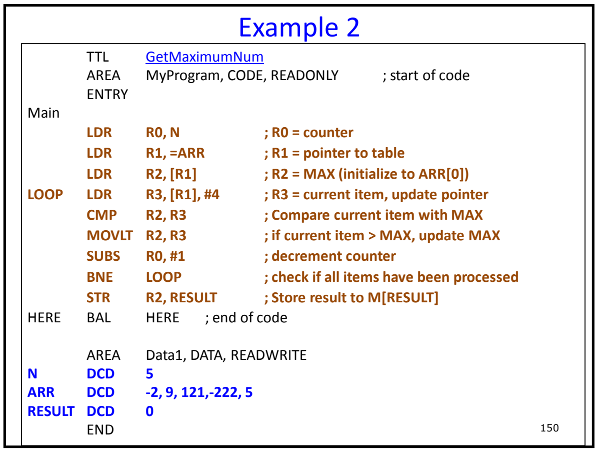
## Example 2

```
RESULT = ARR[0];              // R2 ← MAX (intialize to 1st item)
for (int i=N; i>0; i--)       // R0 ← i,
{
    item = ARR[i|;            // R3 ← item
                              // R1 ← pointer of ARR[i]
    if (item > MAX)
        MAX = item;
}
```



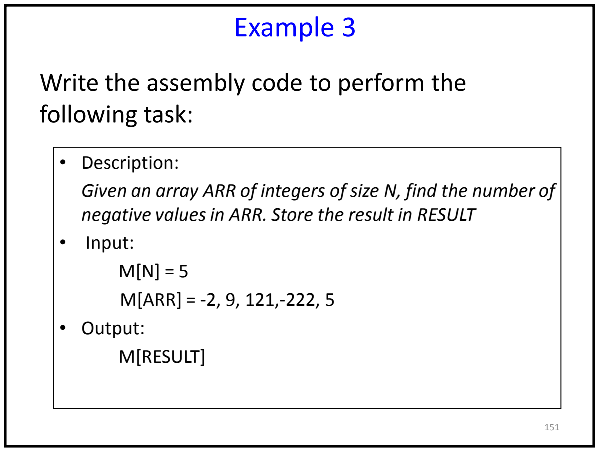
## Example 2

```
i ← N                           ; R0 to store counter
pointer ← ARR                   ; R1 to point to current item in table
MAX ← M[pointer]                ; R2 to store the maximum number
REPEAT
        ITEM ← M[pointer]       ; R3 to store current item
        pointer ← pointer + 4   ; update pointer to next item
        IF ITEM > MAX
            MAX ← ITEM          ; new maximum value if ITEM > MAX
        END IF
        i ← i – 1               ; decrement counter
Until i = 0
M[RESULT] = MAX
```



## Example 2

```
        TTL     GetMaximumNum
        AREA    MyProgram, CODE, READONLY       ; start of code
        ENTRY
Main
        LDR     R0, N                   ; R0 = counter
        LDR     R1, =ARR                ; R1 = pointer to table
        LDR     R2, [R1]                ; R2 = MAX (initialize to ARR[0])
LOOP    LDR     R3, [R1], #4            ; R3 = current item, update pointer
        CMP     R2, R3                  ; Compare current item with MAX
        MOVLT   R2, R3                  ; if current item > MAX, update MAX
        SUBS    R0, #1                  ; decrement counter
        BNE     LOOP                    ; check if all items have been processed
        STR     R2, RESULT              ; Store result to M[RESULT]
HERE    BAL     HERE    ; end of code

        AREA    Data1, DATA, READWRITE
N       DCD     5
ARR     DCD     -2, 9, 121,-222, 5
RESULT  DCD     0
        END
```

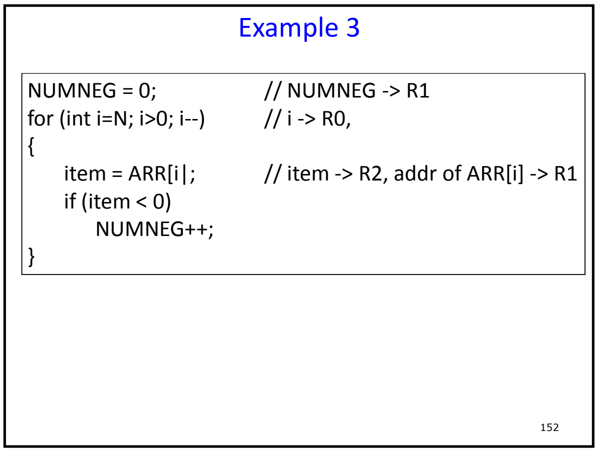


## Example 3

Write the assembly code to perform the following task:

- Description:
  *Given an array ARR of integers of size N, find the number of negative values in ARR. Store the result in RESULT*
- Input:
  M[N] = 5
  M[ARR] = -2, 9, 121,-222, 5
- Output:
  M[RESULT]

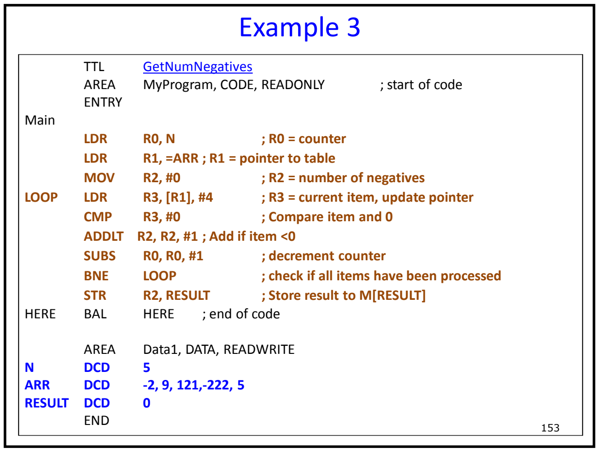
151



## Example 3

```
NUMNEG = 0;                 // NUMNEG -> R1
for (int i=N; i>0; i--)     // i -> R0,
{
    item = ARR[i|;          // item -> R2, addr of ARR[i] -> R1
    if (item < 0)
        NUMNEG++;
}
```

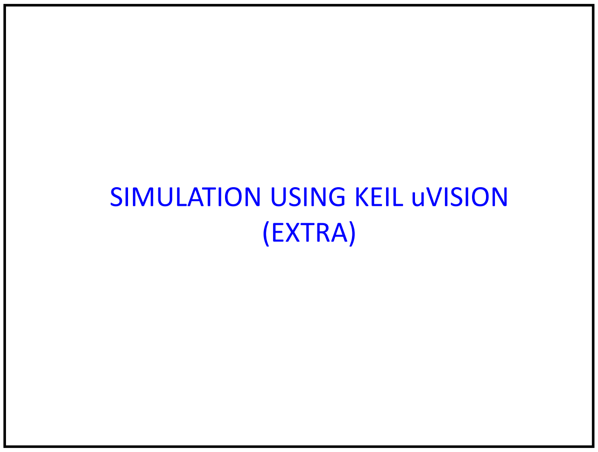
152



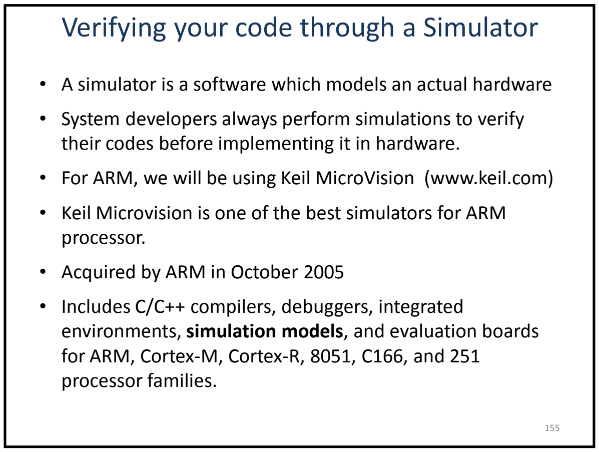
## Example 3

```
        TTL     GetNumNegatives
        AREA    MyProgram, CODE, READONLY           ; start of code
        ENTRY
Main
        LDR     R0, N                   ; R0 = counter
        LDR     R1, =ARR ; R1 = pointer to table
        MOV     R2, #0                  ; R2 = number of negatives
LOOP    LDR     R3, [R1], #4            ; R3 = current item, update pointer
        CMP     R3, #0                  ; Compare item and 0
        ADDLT   R2, R2, #1 ; Add if item <0
        SUBS    R0, R0, #1              ; decrement counter
        BNE     LOOP                    ; check if all items have been processed
        STR     R2, RESULT              ; Store result to M[RESULT]
HERE    BAL     HERE      ; end of code

        AREA    Data1, DATA, READWRITE
N       DCD     5
ARR     DCD     -2, 9, 121,-222, 5
RESULT  DCD     0
        END
```

153



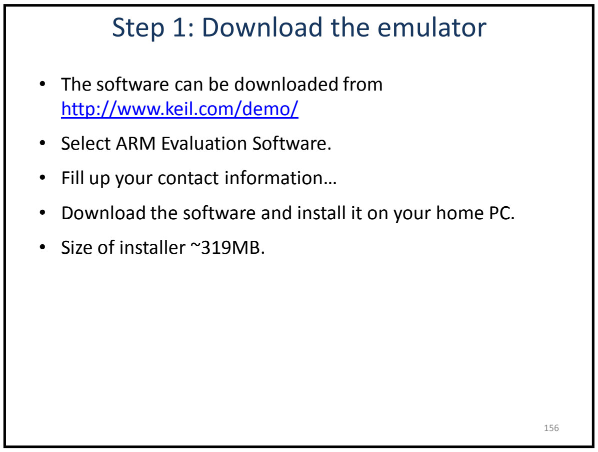
## SIMULATION USING KEIL uVISION (EXTRA)



## Verifying your code through a Simulator

- A simulator is a software which models an actual hardware
- System developers always perform simulations to verify their codes before implementing it in hardware.
- For ARM, we will be using Keil MicroVision (www.keil.com)
- Keil Microvision is one of the best simulators for ARM processor.
- Acquired by ARM in October 2005
- Includes C/C++ compilers, debuggers, integrated environments, **simulation models**, and evaluation boards for ARM, Cortex-M, Cortex-R, 8051, C166, and 251 processor families.

155



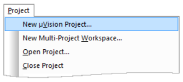
## Step 1: Download the emulator

- The software can be downloaded from http://www.keil.com/demo/
- Select ARM Evaluation Software.
- Fill up your contact information…
- Download the software and install it on your home PC.
- Size of installer ~319MB.

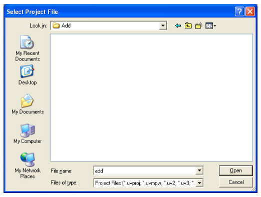
156

## Step 2: Create your project

- Select **Project - New Project** from the µVision4 menu.
  This opens a standard Windows dialog, which prompts you for the new project file name.



- Create a new folder ***Add*** in your preferred destination directory.
- Switch to the new folder and type the project name **Add**. µVision4 automatically adds the extension **.uvproj**.
- Click **Save**.



Look in: Add

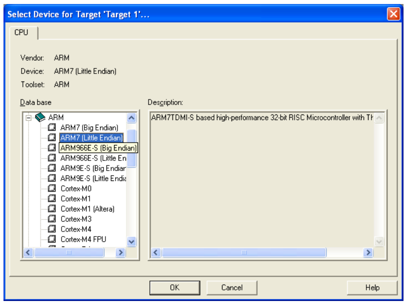
File name: add

Files of type: Project Files (*.uvproj; *.uvmpw; *.uv2; *.uv3; *.

Note: *Use a separate folder for each project.*

## Step 3: Select Device

Select the microcontroller you use. For our tutorials, choose ARM > ARM 7 (Big Endian)



Vendor: ARM

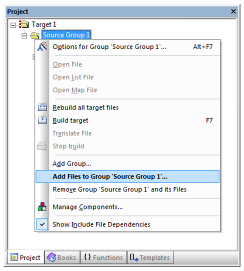
Device: ARM7 (Little Endian)

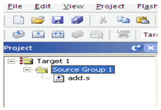
Toolset: ARM

Description: ARM7TDMI-S based high-performance 32-bit RISC Microcontroller with Tl

## Step 4: Write your assembly code

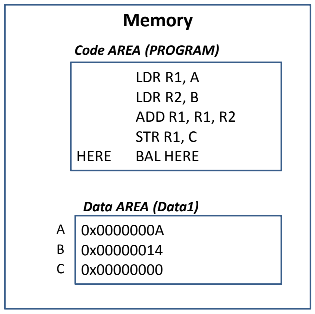
- Create your file: Go to **File - New** menu. This opens an empty editor window where you can enter your source code. µVision4 enables color syntax highlighting, when the file is saved with the extension ***.S**, ***.C** or ***.CPP**.
- Save the file using **File - Save As...**, and name it, e.g. , **add.s**.  The extension .s is an assembly code file.
- Write the code in the text editor
- Add the file to your project. Invoke the **Context Menu** of the **Source Group 1** in the **Project Window** (shown in the left Figure)
- The navigation panel should look like this





## Step 5: Setting the environment

- To run the program, your program needs to be loaded into the main memory.
- Different locations may be assigned for the code and data area, depending on your setup.
- In the simulation environment of uVision, this can be set.
- In a physical environment, this is determined by the hardware setup.



**Memory**

*Code AREA (PROGRAM)*

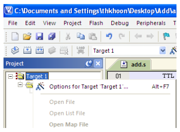
```
        LDR R1, A
        LDR R2, B
        ADD R1, R1, R2
        STR R1, C
HERE    BAL HERE
```

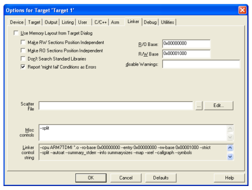
*Data AREA (Data1)*

```
A   0x0000000A
B   0x00000014
C   0x00000000
```

## Step 5: Setting the environment

- Right-click on ***Target 1*** in the navigational panel and select *'**Options for target 'Target 1**"*
- The options menu box will pop up. Goto ***Linker***. For our classes, we shall follow the following setup for all our tutorials.
- Use separate regions for the code and data.
  - Set **R/O Base** to **0x00000000** to define the ***code area***
  - Set **R/W Base** to **0x00001000** to define the ***data area***
  - Put an extra parameter ***--split*** at *misc controls* (double hyphen)





☐ Use Memory Layout from Target Dialog

☐ Make RW Sections Position Independent

☐ Make RO Sections Position Independent

☐ Don't Search Standard Libraries

☑ Report 'might fail' Conditions as Errors

R/O Base: 0x00000000

R/W Base: 0x00001000

disable Warnings:

Misc controls: --split

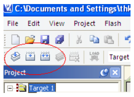
Linker control string:

```
--cpu ARM7TDMI *.o --ro-base 0x00000000 --entry 0x00000000 --rw-base 0x00001000 --strict
--split --autoat --summary_stderr --info summarysizes --map --xref --callgraph --symbols
```

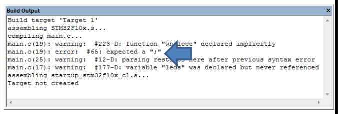
*Source:* http://www.keil.com/support/man/docs/armlink/armlink_Cacbdbbc.htm

## Step 6: Assembling your program

- Assemble and link the source files of the application by clicking on one of the build-buttons located in the ***Build Toolbar.***

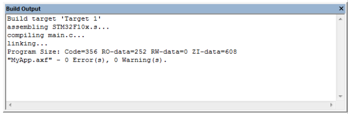


- If there are errors, they will be displayed in the Build Output Window. Double-click to jump to the error line. Correct the error and recompile.



```
Build target 'Target 1'
assembling STM32F10x.s...
compiling main.c...
main.c(19): warning:  #223-D: function "whilcce" declared implicitly
main.c(19): error:  #65: expected a ";"
main.c(25): warning:  #12-D: parsing restores here after previous syntax error
main.c(17): warning:  #177-D: variable "leds" was declared but never referenced
assembling startup_stm32f10x_cl.s...
Target not created
```

***With Error***



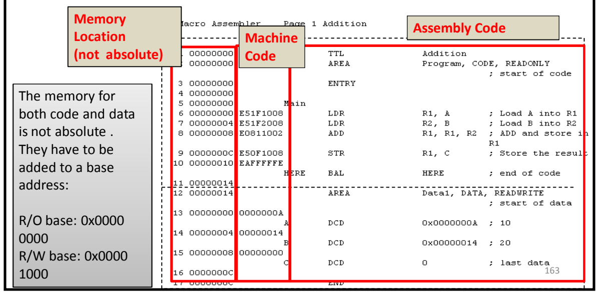
```
Build target 'Target 1'
assembling STM32F10x.s...
compiling main.c...
linking...
Program Size: Code=356 RO-data=252 RW-data=0 ZI-data=608
"MyApp.axf" - 0 Error(s), 0 Warning(s).
```

***Without Error***

***Important: Remember to re-build your program after you make changes to your source code!!!***

## Debugging Using the .O and .LST File

• After assembling your program, three important file will be created in your project directory:
**a).o file** which stores the machine code/binary to be stored in the memory
**b).lst file** is a file which shows the result of assembled machine code arranged line by line with your assembly code. This file is created for your debugging purposes only.



Memory Location (not absolute)

Machine Code

Assembly Code

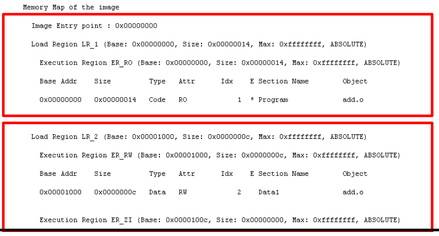
The memory for both code and data is not absolute . They have to be added to a base address:

R/O base: 0x0000 0000
R/W base: 0x0000 1000

```
Macro Assembler    Page 1 Addition

 1 00000000                 TTL          Addition
 2 00000000                 AREA         Program, CODE, READONLY
                                                      ; start of code
 3 00000000                 ENTRY
 4 00000000
 5 00000000          Main
 6 00000000 E51F1008        LDR          R1, A        ; Load A into R1
 7 00000004 E51F2008        LDR          R2, B        ; Load B into R2
 8 00000008 E0811002        ADD          R1, R1, R2   ; ADD and store in
                                                      R1
 9 0000000C E50F1008        STR          R1, C        ; Store the result
10 00000010 EAFFFFFE
                     HERE   BAL          HERE         ; end of code
11 00000014
12 00000014                 AREA         Data1, DATA, READWRITE
                                                      ; start of data
13 00000000 0000000A
                     A      DCD          0x0000000A   ; 10
14 00000004 00000014
                     B      DCD          0x00000014   ; 20
15 00000008 00000000
                     C      DCD          0            ; last data
16 0000000C
17 0000000C                 END
```

## MAP File

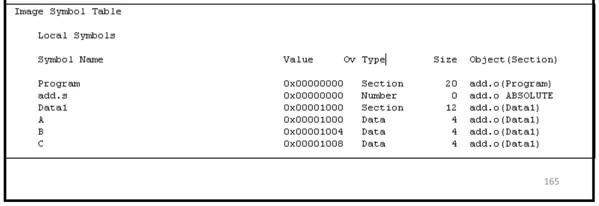
• After assembling your program, three important file will be created in your project directory:
**a).o file** which stores the machine code/binary to be stored in the memory
**b).lst file** is a file which shows the result of assembled machine code arranged line by line with your assembly code. This file is created for your debugging purposes only.
**c).map file** which shows the absolute address of the code and data area.
- ***Memory map*** specifies where the code and data section is loaded in the memory



```
Memory Map of the image

  Image Entry point : 0x00000000

  Load Region LR_1 (Base: 0x00000000, Size: 0x00000014, Max: 0xffffffff, ABSOLUTE)

    Execution Region ER_RO (Base: 0x00000000, Size: 0x00000014, Max: 0xffffffff, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00000000   0x00000014   Code   RO           1   * Program             add.o

  Load Region LR_2 (Base: 0x00001000, Size: 0x0000000c, Max: 0xffffffff, ABSOLUTE)

    Execution Region ER_RW (Base: 0x00001000, Size: 0x0000000c, Max: 0xffffffff, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00001000   0x0000000c   Data   RW           2     Data1               add.o


    Execution Region ER_ZI (Base: 0x0000100c, Size: 0x00000000, Max: 0xffffffff, ABSOLUTE)
```

## MAP File

• After assembling your program, three important file will be created in your project directory:
**a).o file** which stores the machine code/binary to be stored in the memory
**b).lst file** is a file which shows the result of assembled machine code arranged line by line with your assembly code. This file is created for your debugging purposes only.
**c).map file** which shows the absolute address of the code and data area.
- ***Memory map*** specifies where the code and data section is loaded in the memory
- ***Symbol table*** specifies the absolute address of each variable

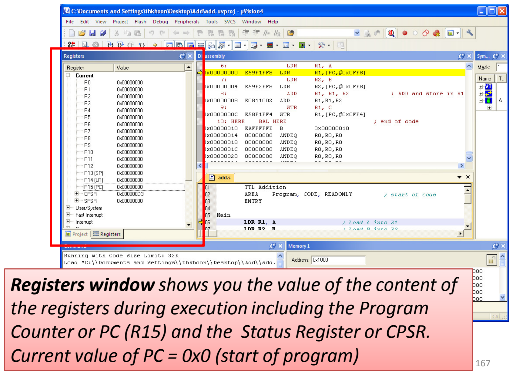


```
Image Symbol Table

    Local Symbols

    Symbol Name                              Value      Ov Type        Size  Object(Section)

    Program                                  0x00000000   Section        20  add.o(Program)
    add.s                                    0x00000000   Number          0  add.o ABSOLUTE
    Data1                                    0x00001000   Section        12  add.o(Data1)
    A                                        0x00001000   Data            4  add.o(Data1)
    B                                        0x00001004   Data            4  add.o(Data1)
    C                                        0x00001008   Data            4  add.o(Data1)
```

## Step 7: Running and Debugging

- ***Debugging*** is one of the most important skill you need to acquire when writing programs. If you do not know how to debug your program, you have not really master programming.
- The debugging windows allows you to do the following
  1) ***Step through*** your program line-by-line. From here, we can check if the execution flow of your code is according to what you expect the program to do. If it doesn't, you can pinpoint what is wrong with your code.
  2) Monitor ***the values of the registers***. From here, we can monitor if the register has been updated correctly.
  3) Monitor the ***values of the specified memory***. From here, we can monitor if the memory is being updated/loaded correctly.
- Click **Debug > Start/Stop Debug Session** (or Ctrl-F5) to start debugging

## Step 7: Running and Debugging


C:\Documents and Settings\thkhoon\Desktop\Add\add.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Registers
Disassembly
add.s
Sym...
Mask: *
Name T..
Project Registers
Command
Memory 1
Address: 0x1000

| Register | Value |
|---|---|
| Current | |
| R0 | 0x00000000 |
| R1 | 0x00000000 |
| R2 | 0x00000000 |
| R3 | 0x00000000 |
| R4 | 0x00000000 |
| R5 | 0x00000000 |
| R6 | 0x00000000 |
| R7 | 0x00000000 |
| R8 | 0x00000000 |
| R9 | 0x00000000 |
| R10 | 0x00000000 |
| R11 | 0x00000000 |
| R12 | 0x00000000 |
| R13 (SP) | 0x00000000 |
| R14 (LR) | 0x00000000 |
| R15 (PC) | 0x00000000 |
| CPSR | 0x000000D3 |
| SPSR | 0x00000000 |
| User/System | |
| Fast Interrupt | |
| Interrupt | |

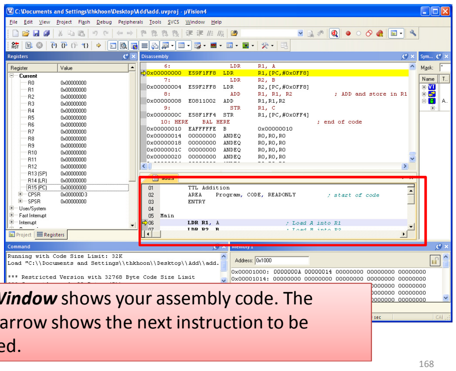
```
     6:                 LDR      R1, A
0x00000000  E59F1FF8  LDR      R1,[PC,#0x0FF8]
     7:                 LDR      R2, B
0x00000004  E59F2FF8  LDR      R2,[PC,#0x0FF8]
     8:                 ADD      R1, R1, R2          ; ADD and store in R1
0x00000008  E0811002  ADD      R1,R1,R2
     9:                 STR      R1, C
0x0000000C  E58F1FF4  STR      R1,[PC,#0x0FF4]
    10: HERE    BAL HERE                       ; end of code
0x00000010  EAFFFFFE  B        0x00000010
0x00000014  00000000  ANDEQ    R0,R0,R0
0x00000018  00000000  ANDEQ    R0,R0,R0
0x0000001C  00000000  ANDEQ    R0,R0,R0
0x00000020  00000000  ANDEQ    R0,R0,R0
```

```
01          TTL Addition
02          AREA    Program, CODE, READONLY        ; start of code
03          ENTRY
04
05 Main
06          LDR R1, A                     ; Load A into R1
```

```
Running with Code Size Limit: 32K
Load "C:\\Documents and Settings\\thkhoon\\Desktop\\Add\\add.
```

***Registers window*** *shows you the value of the content of the registers during execution including the Program Counter or PC (R15) and the Status Register or CPSR. Current value of PC = 0x0 (start of program)*

## Step 7: Running and Debugging


C:\Documents and Settings\thkhoon\Desktop\Add\add.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Registers
Disassembly
add.s
Sym...
Mask: *
Name T..
Project Registers
Command
Memory 1
Address: 0x1000

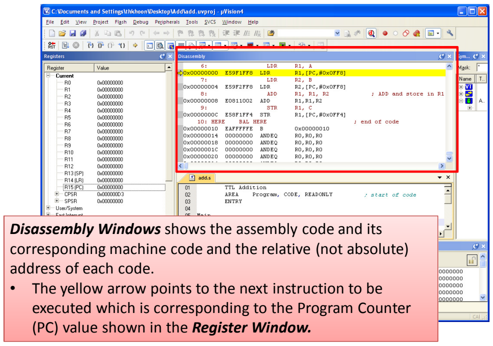
```
01          TTL Addition
02          AREA    Program, CODE, READONLY        ; start of code
03          ENTRY
04
05 Main
06          LDR R1, A                     ; Load A into R1
```

```
Running with Code Size Limit: 32K
Load "C:\\Documents and Settings\\thkhoon\\Desktop\\Add\\add.

*** Restricted Version with 32768 Byte Code Size Limit
```

```
0x00001000: 0000000A 00000014 00000000 00000000 00000000
0x00001014: 00000000 00000000 00000000 00000000 00000000
```

***Code Window*** shows your assembly code. The yellow arrow shows the next instruction to be executed.

## Step 7: Running and Debugging



C:\Documents and Settings\thkhoon\Desktop\Add\add.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Registers
Disassembly
add.s

| Register | Value |
|---|---|
| Current | |
| R0 | 0x00000000 |
| R1 | 0x00000000 |
| R2 | 0x00000000 |
| R3 | 0x00000000 |
| R4 | 0x00000000 |
| R5 | 0x00000000 |
| R6 | 0x00000000 |
| R7 | 0x00000000 |
| R8 | 0x00000000 |
| R9 | 0x00000000 |
| R10 | 0x00000000 |
| R11 | 0x00000000 |
| R12 | 0x00000000 |
| R13 (SP) | 0x00000000 |
| R14 (LR) | 0x00000000 |
| R15 (PC) | 0x00000000 |
| CPSR | 0x000000D3 |
| SPSR | 0x00000000 |
| User/System | |

```
       6:                  LDR      R1, A
0x00000000  E59F1FF8  LDR      R1,[PC,#0x0FF8]
       7:                  LDR      R2, B
0x00000004  E59F2FF8  LDR      R2,[PC,#0x0FF8]
       8:                  ADD      R1, R1, R2          ; ADD and store in R1
0x00000008  E0811002  ADD      R1,R1,R2
       9:                  STR      R1, C
0x0000000C  E58F1FF4  STR      R1,[PC,#0x0FF4]
    10: HERE    BAL HERE                       ; end of code
0x00000010  EAFFFFFE  B        0x00000010
0x00000014  00000000  ANDEQ    R0,R0,R0
0x00000018  00000000  ANDEQ    R0,R0,R0
0x0000001C  00000000  ANDEQ    R0,R0,R0
0x00000020  00000000  ANDEQ    R0,R0,R0
```

```
01          TTL Addition
02          AREA    Program, CODE, READONLY          ; start of code
03          ENTRY
04
```

***Disassembly Windows*** shows the assembly code and its corresponding machine code and the relative (not absolute) address of each code.

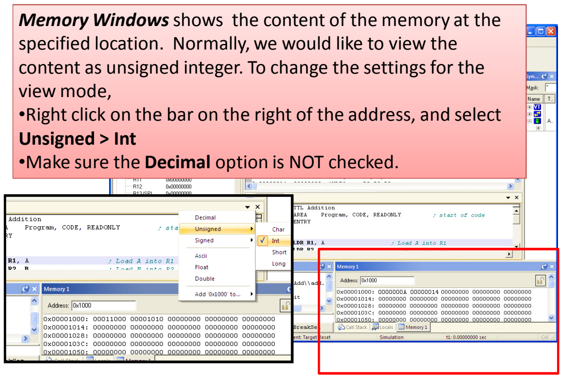
- The yellow arrow points to the next instruction to be executed which is corresponding to the Program Counter (PC) value shown in the ***Register Window.***

## Step 7: Running and Debugging

***Memory Windows*** shows the content of the memory at the specified location. Normally, we would like to view the content as unsigned integer. To change the settings for the view mode,

•Right click on the bar on the right of the address, and select **Unsigned > Int**

•Make sure the **Decimal** option is NOT checked.



```
          TTL Addition
          AREA    Program, CODE, READONLY          ; start of code
          ENTRY

          LDR R1, A                   ; Load A into R1
```

Address: 0x1000

```
0x00001000: 00011000 00001010 00000000 00000000 00000000
0x00001014: 00000000 00000000 00000000 00000000 00000000
0x00001028: 00000000 00000000 00000000 00000000 00000000
0x0000103C: 00000000 00000000 00000000 00000000 00000000
0x00001050: 00000000 00000000 00000000 00000000 00000000
```

Address: 0x1000

```
0x00001000: 0000000A 00000014 00000000 00000000 00000000
0x00001014: 00000000 00000000 00000000 00000000 00000000
0x00001028: 00000000 00000000 00000000 00000000 00000000
0x0000103C: 00000000 00000000 00000000 00000000 00000000
0x00001050: 00000000 00000000 00000000 00000000 00000000
```

## Step 7: Running and Debugging

***Memory Windows***
Type the address you are interested to observe its content

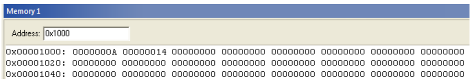
**Assembly code (.s)**

```
        ...
        AREA    Data1, DATA,
READWRITE               ; start of data
A       DCD     0x0000 000A
        DCD     0x0000 0014
C       DCD     0
```

**From the map file (.map):**

```
Image Symbol Table

    Local Symbols

    Symbol Name                              Value

    Program                                  0x00000000
    add.s                                    0x00000000
    Data1                                    0x00001000
    A                                        0x00001000
    B                                        0x00001004
    C                                        0x00001008
```

**Memory content:**

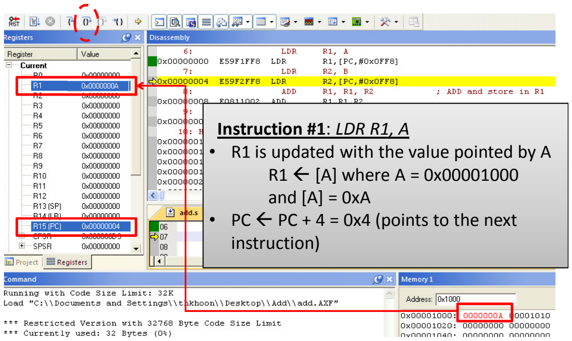


Address: 0x1000

```
0x00001000: 0000000A 00000014 00000000 00000000 00000000 00000000 00000000 00000000
0x00001020: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x00001040: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

## Step 7: Running and Debugging

- To run an instruction, **press F10** or press the Step-Into/Step-Through Button. *Yellow line indicates next instruction to run.*



R1 0x0000000A

R15 (PC) 0x00000004

```
       6:                  LDR      R1, A
0x00000000  E59F1FF8  LDR      R1,[PC,#0x0FF8]
       7:                  LDR      R2, B
0x00000004  E59F2FF8  LDR      R2,[PC,#0x0FF8]
       8:                  ADD      R1, R1, R2          ; ADD and store in R1
```

```
Running with Code Size Limit: 32K
Load "C:\\Documents and Settings\\thkhoon\\Desktop\\Add\\add.AXF"

*** Restricted Version with 32768 Byte Code Size Limit
*** Currently used: 32 Bytes (0%)
```

Address: 0x1000

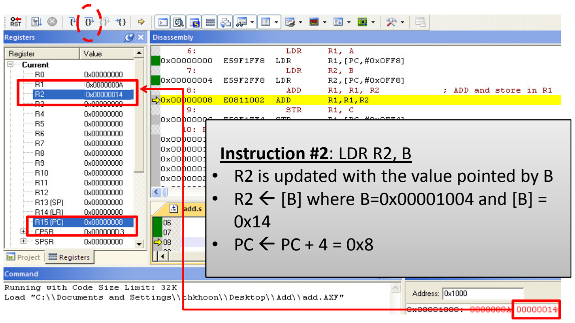
```
0x00001000: 0000000A 00001010
0x00001020: 00000000 00000000
```

**<u>Instruction #1</u>**: *<u>LDR R1, A</u>*

- R1 is updated with the value pointed by A
  R1 ← [A] where A = 0x00001000
  and [A] = 0xA
- PC ← PC + 4 = 0x4 (points to the next instruction)

## Step 7: Running and Debugging

- To run an instruction, **press F10** or press the Step-Into/Step-Through Button



R1 0x0000000A

R2 0x00000014

R15 (PC) 0x00000008

CPSR 0x000000D3

```
       6:                  LDR      R1, A
0x00000000  E59F1FF8  LDR      R1,[PC,#0x0FF8]
       7:                  LDR      R2, B
0x00000004  E59F2FF8  LDR      R2,[PC,#0x0FF8]
       8:                  ADD      R1, R1, R2          ; ADD and store in R1
0x00000008  E0811002  ADD      R1,R1,R2
       9:                  STR      R1, C
```

```
Running with Code Size Limit: 32K
Load "C:\\Documents and Settings\\thkhoon\\Desktop\\Add\\add.AXF"
```

Address: 0x1000

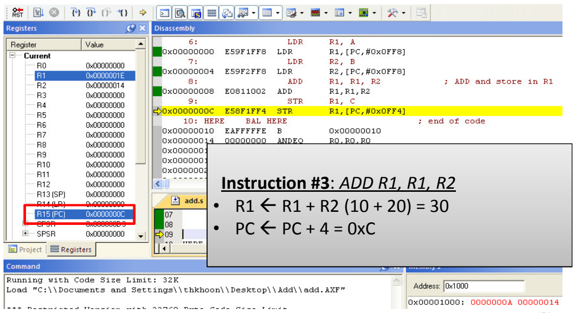
```
0x00001000: 0000000A 00000014
```

**<u>Instruction #2</u>**: <u>LDR R2, B</u>

- R2 is updated with the value pointed by B
- R2 ← [B] where B=0x00001004 and [B] = 0x14
- PC ← PC + 4 = 0x8

## Step 7: Running and Debugging

- To run an instruction, **press F10** or press the Step-Into/Step-Through Button



| Register | Value |
|---|---|
| Current | |
| R0 | 0x00000000 |
| R1 | 0x0000001E |
| R2 | 0x00000014 |
| R3 | 0x00000000 |
| R4 | 0x00000000 |
| R5 | 0x00000000 |
| R6 | 0x00000000 |
| R7 | 0x00000000 |
| R8 | 0x00000000 |
| R9 | 0x00000000 |
| R10 | 0x00000000 |
| R11 | 0x00000000 |
| R12 | 0x00000000 |
| R13 (SP) | 0x00000000 |
| R15 (PC) | 0x0000000C |
| SPSR | 0x00000000 |

```
       6:                  LDR      R1, A
0x00000000  E59F1FF8  LDR      R1,[PC,#0x0FF8]
       7:                  LDR      R2, B
0x00000004  E59F2FF8  LDR      R2,[PC,#0x0FF8]
       8:                  ADD      R1, R1, R2          ; ADD and store in R1
0x00000008  E0811002  ADD      R1,R1,R2
       9:                  STR      R1, C
0x0000000C  E58F1FF4  STR      R1,[PC,#0x0FF4]
    10: HERE    BAL HERE                       ; end of code
0x00000010  EAFFFFFE  B        0x00000010
0x00000014  00000000  ANDEQ    R0,R0,R0
```

```
Running with Code Size Limit: 32K
Load "C:\\Documents and Settings\\thkhoon\\Desktop\\Add\\add.AXF"
```

Address: 0x1000

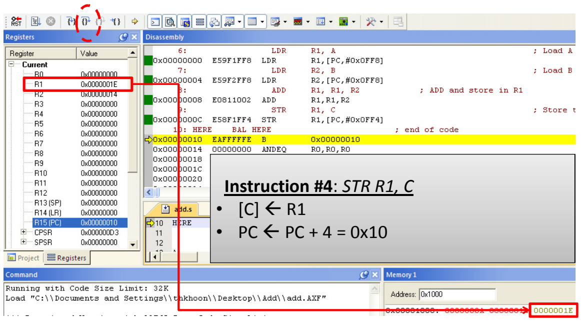
```
0x00001000: 0000000A 00000014
```

**<u>Instruction #3</u>**: *<u>ADD R1, R1, R2</u>*

- R1 ← R1 + R2 (10 + 20) = 30
- PC ← PC + 4 = 0xC

## Step 7: Running and Debugging

- To run an instruction, **press F10** or press the Step-Into/Step-Through Button



| Register | Value |
|---|---|
| Current | |
| R0 | 0x00000000 |
| R1 | 0x0000001E |
| R2 | 0x00000014 |
| R3 | 0x00000000 |
| R4 | 0x00000000 |
| R5 | 0x00000000 |
| R6 | 0x00000000 |
| R7 | 0x00000000 |
| R8 | 0x00000000 |
| R9 | 0x00000000 |
| R10 | 0x00000000 |
| R11 | 0x00000000 |
| R12 | 0x00000000 |
| R13 (SP) | 0x00000000 |
| R14 (LR) | 0x00000000 |
| R15 (PC) | 0x00000010 |
| CPSR | 0x000000D3 |
| SPSR | 0x00000000 |

```
     6:                  LDR      R1, A                              ; Load A
0x00000000  E59F1FF8  LDR      R1,[PC,#0x0FF8]
     7:                  LDR      R2, B                              ; Load B
0x00000004  E59F2FF8  LDR      R2,[PC,#0x0FF8]
     8:                  ADD      R1, R1, R2           ; ADD and store in R1
0x00000008  E0811002  ADD      R1,R1,R2
     9:                  STR      R1, C                              ; Store t
0x0000000C  E58F1FF4  STR      R1,[PC,#0x0FF4]
    10: HERE    BAL HERE                          ; end of code
0x00000010  EAFFFFFE  B        0x00000010
0x00000014  00000000  ANDEQ    R0,R0,R0
0x00000018
0x0000001C
0x00000020
```

```
10  HERE
11
12
```

**Instruction #4**: *STR R1, C*

- [C] ← R1
- PC ← PC + 4 = 0x10

```
Running with Code Size Limit: 32K
Load "C:\\Documents and Settings\\thkhoon\\Desktop\\Add\\add.AXF"
```

Address: 0x1000

```
0x00001000: 0000000A 00000014 0000001E
```