Embedded Systems

Ch 12A ARM Assembly Language

Byung Kook Kim
Dept of EECS
Korea Advanced Institute of Science and Technology

Overview

- *1. Introduction*
- *2. Data Processing Instructions*
- *3. Data Transfer Instructions*
- *4. Control Flow Instructions*
- *5. Writing Simple Assembly Language Programs*

- References
  - Steve Furber, “ARM System-on-chip architecture”, Second Edition, Addison Wesley, 2000.

1. Introduction

- **ARM processor programming**
  - C, C++, or assembly language

- **Assembly language programming**
  - Think at the level of individual machine instruction
  - **Assembler**: computer program converting assembly language programs into machine language programs
    - Binary-level instruction encoding
  - Level
    - User level programming
    - System level programming: Next chapter
  - Instruction size
    - 32-bit ARM assembly language
    - 16-bit ARM Thumb instructions

Introduction (II)

- **ARM data types**
  - 8-bit signed and unsigned bytes
  - 16-bit signed and unsigned half-words aligned on 2-byte boundaries
    - Some older ARM processors do not have half-word and signed byte support
  - 32-bit signed and unsigned words aligned on 4-byte boundaries

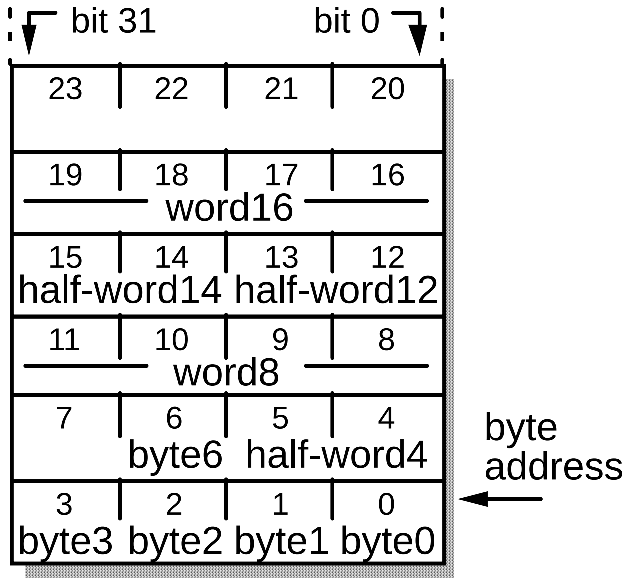
  - ARM instruction: 32 bits. Must be word-aligned
  - Thumb instruction: 16 bits. Must be aligned on 2-byte boundaries

  - Internal ARM operations
    - 32-bit operands
    - Byte loaded: Zero or sign-extended. Treated as a 32-bit value.

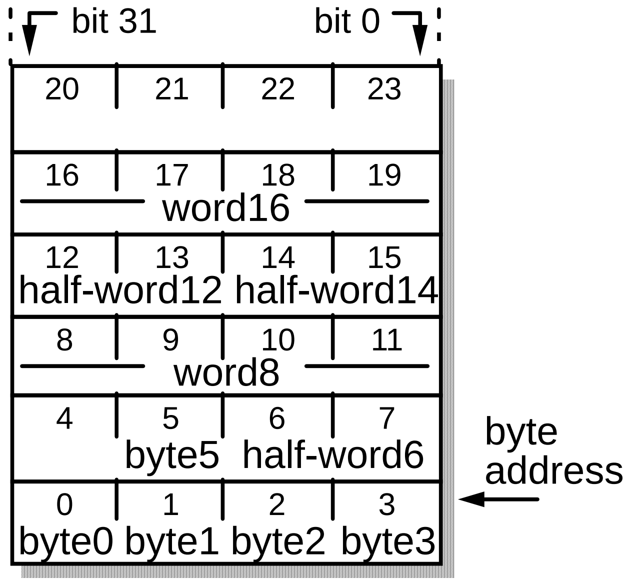
  - ARM coprocessor: may support floating-point values.

Introduction (III)

- **Memory organization**
  - Storing words in a byte-addressed memory
    - **Little-endian**: Least significant byte first. Intel. ARM default.
    - **Big-endian**: Most significant byte first. Motorola.



(a) Little-endian memory organization



(b) Big-endian memory organization

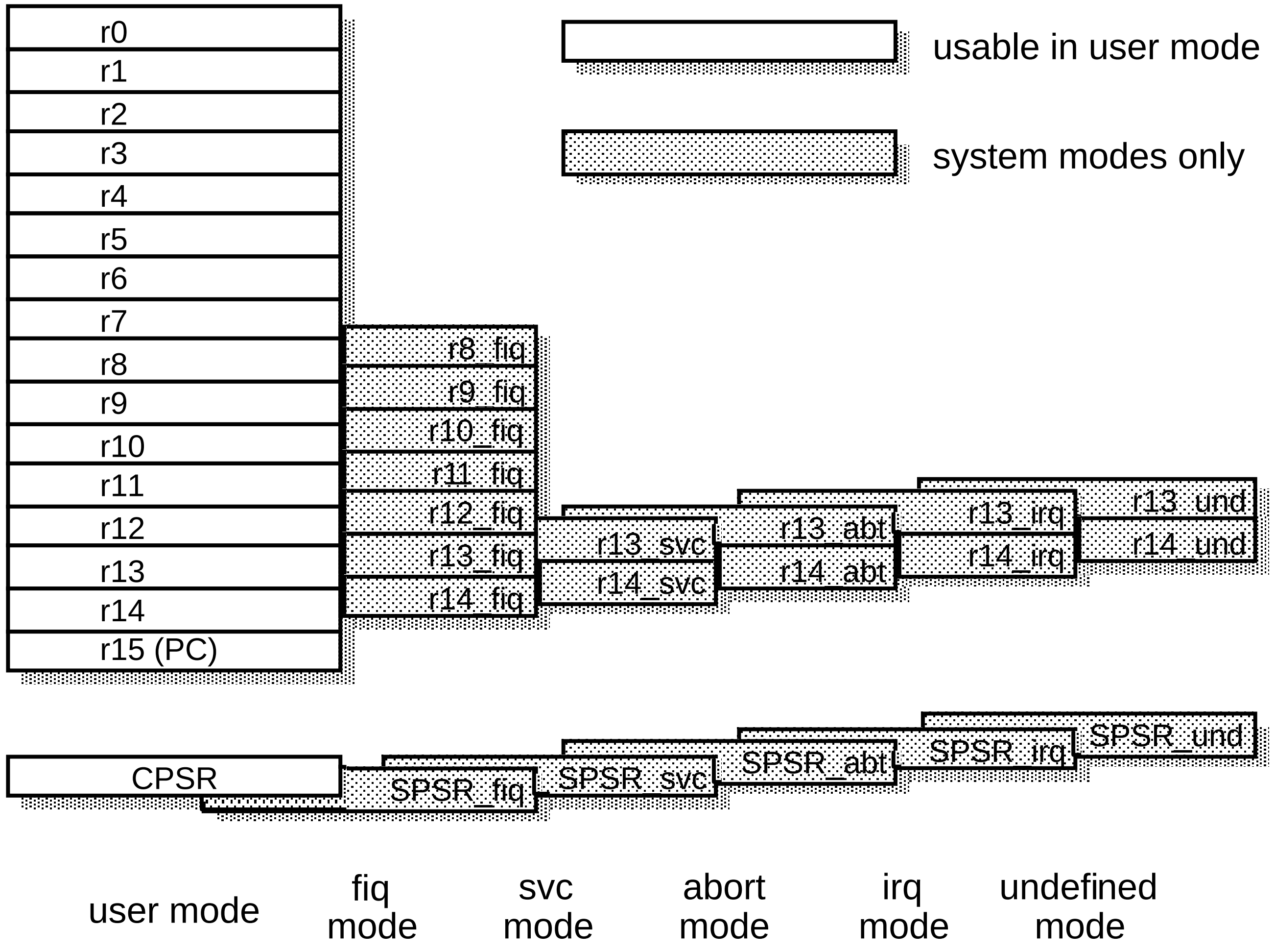
Introduction (IV)

- **Privileged modes**
  - Used to handle exceptions and supervisor calls (software interrupts)
  - Current operating mode: CPSR [4:0]

CPSR[4:0]	Mode	Use	Registers
10000	User	Normal user code	user
10001	FIQ	Processing fast interrupts	_fiq
10010	IRQ	Processing standard interrupts	_irq
10011	SVC	Processing software interrupts (SWIs)	_svc
10111	Abort	Processing memory faults	_abt
11011	Undef	Handling undefined instruction traps	_und
11111	System	Running privileged operating system tasks	user

  - Shaded registers replace the corresponding user registers
  - Current SPSR (Saved Program Status Register) also becomes available.

Introduction (V)



Introduction (VI)

- **Privileged modes (cont'd)**
  - Can only be entered through controlled mechanisms
  - Allow a fully protected operating system to be built
  - Can be used to give a weaker level of protection useful for trapping errant software.

  - SPSRs
    - Used to save the state of CSPR when the privileged mode is entered
    - CSPR restored when exit (resume user program)

    - Re-entrant privileged software: the CSPR must be copied into a general register and saved.

2. Data Processing Instructions

- Arithmetic and logic operations on data values in registers
  - Two operands and one result

- **Rules**
  - All operands are 32 bits wide and come from registers or are specified as literals in the instruction itself
  - The result, if there is one, is 32 bits wide and is placed in a register
    - Exception: Long multiply instruction produces a 64-bit result
  - Each of the operand registers and the result register are independently specified in the instruction (3-address format).

Data Processing Instructions (II)

- **Simple register operands**
  - Format: OP_code dest, src1, src2
  - Ex) ADD r0, r1, r2 ; r0 := r1 + r2
    - Semicolon (;): Comment (to the right of it)
      - Easier reading and understanding
    - May produce a carry output, overflow
      - Stored in N, Z, C, and V flags in CSPR.
  - Arithmetic operations
    - ADD r0, r1, r2 ; r0 := r1 + r2
    - ADC r0, r1, r2 ; r0 := r1 + r2 + C. Add with carry
    - SUB r0, r1, r2 ; r0 := r1 – r2
    - SBC r0, r1, r2 ; r0 := r1 – r2 + C – 1. Subtract with carry
    - RSB r0, r1, r2 ; r0 := r2 – r1
    - RSC r0, r1, r2 ; r0 := r2 – r1 + C – 1. Rev sub with carry.

Data Processing Instructions (III)

- **Simple register operands (Cont'd)**
  - Bitwise logical operations

AND r0, r1, r2	; r0 := r1 and r2
ORR r0, r1, r2	; r0 := r1 or r2
EOR r0, r1, r2	; r0 := r1 xor r2. Exclusive or
BIC r0, r1, r2	; r0 := r1 and not r2. Bitwise clear

  - Register movement operations

MOV r0, r2	; r0 := r2
MVN r0, r2	; r0 := not r2. Move not (negated)

  - Compare operations

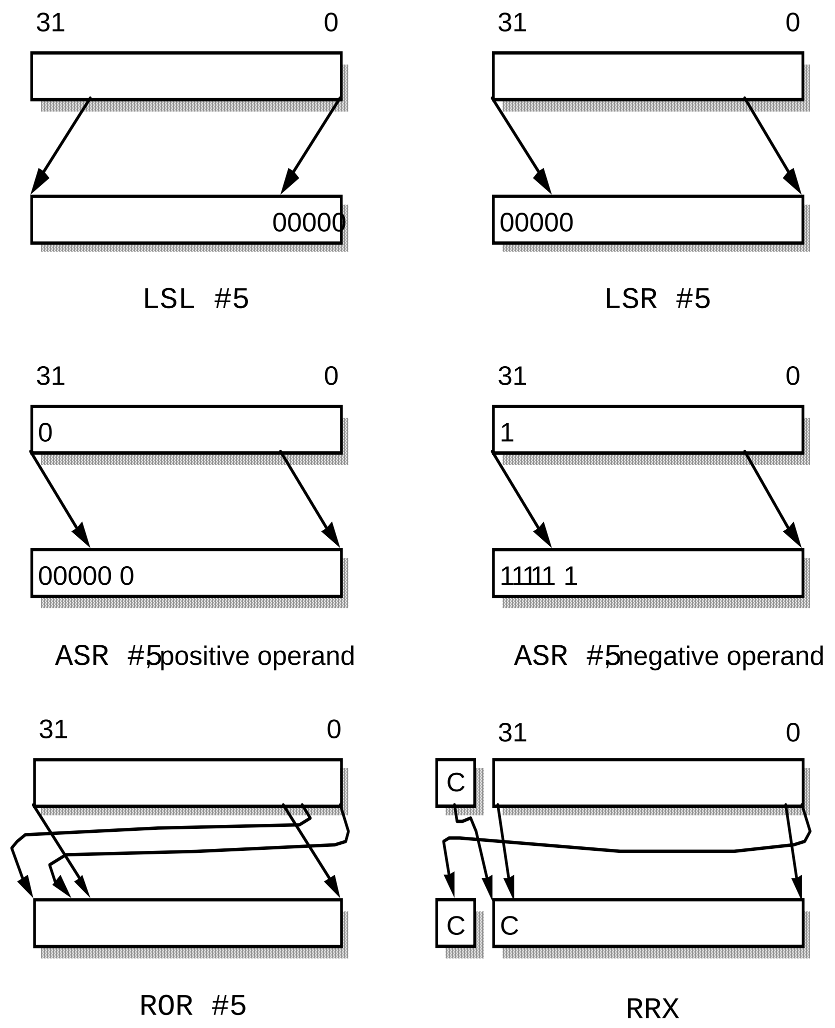
CMP r1, r2	; Set CC on r1 – r2. Compare
CMN r1, r2	; Set CC on r1 + r2. Compare negated
TST r1, r2	; Set CC on r1 and r2
TEQ r1, r2	; Set CC on r1 xor r2. Test equal

Data Processing Instructions (IV)

- **Immediate operands**
  - Constant preceded by ‘#’
    - ADD r3, r3, #1 ; r3 := r3 + 1
  - Hexadecimal constant preceded by ‘&’ after the ‘#’
    - AND r8, r7, #&ff ; r8 := r7[7:0]
  - Valid immediate values
    - Immediate = (0 to 255) x 2^{2n}, 0<=n<=12.

Data Processing Instructions (V)

- **Shift register operands**
  - Second operand shifted before operation
    - ADD r3, r2, r1, LSL #3 ; r3 := r2 + 8 x r1. Logical shift left
      - Single ARM instruction executed in a single clock cycle.
    - Shift value: 0 to 31
  - Shift operations
    - LSL: logical shift left by 0 to 31 places; fill LSB with zeros
    - LSR: Logical shift right by 0 to 31 places; fill MSB with zeros
    - ASL: Arithmetic shift left; synonym for LSL.
    - ASR: Arithmetic shift right by 0 to 31 places; fill MSB with 0/1
    - ROR: Rotate right by 0 to 31 places; LSBs to MSBs
    - RRX: Rotate right extended by 1 place . Operand & C



Data Processing Instructions (VI)

- **Setting the condition codes**
  - Data processing instructions: optional
  - 'S': Suffix of set condition code
    - ADDS r2, r2, r0 ; 32-bit carry out -> C.
    - ADC r3, r3, r1 ; And added into high word. 64-bit add.
  - Comparison instructions: no option
  - Arithmetic operation: sets all flags
  - Logical & move instruction: Set N and Z. Preserve V.

- **Use of condition codes**
  - C flag: as an input to an arithmetic data processing
  - Conditional branch instructions.

Data Processing Instructions (VII)

- **Multiplies**
  - Multiplication
    - MUL r4, r3, r2 ; r4 := (r3 x r2)[31:0]
  - Differences
    - Immediate second operands are not supported
    - The result register must not be the same as the first source register
    - If the 'S' bit is set, the V flag is preserved. C rendered meaningless.
  - Alternative form
    - MLA r4, r3, r2, r1 ; r4 := (r3 x r2 + r1)[31:0]
  - Multiply by const
    - ADD r0, r0, r0, LSL #2 ; r0 := 5 x r0
    - RSB r0, r0, r0, LSL #3 ; r0 := 7 x r0

3. Data Transfer Instructions

- **Basic forms of data transfer instructions**
  - Single register load and store instructions
    - Transfer between register and memory
    - Byte, 16-bit half word, or 32-bit word
  - Multiple register load and store instructions
    - Enable large quantities of data to be transferred more efficiently
    - Used for procedure entry and exit
    - Save and restore workspace registers
    - Copy blocks of data around memory
  - Single register swap instructions
    - Allow a value in a register to be exchanged with a value in memory
    - Implement semaphores to ensure mutual exclusion.

Data Transfer Instructions (II)

- **Register indirect addressing**
  - Use register value as a memory address
    - LDR r0, [r1] ; r0 := mem[r1]
    - STR r0, [r1] ; mem[r1] := r0

- **Initializing an address register**
  - A **base register** within 4K bytes should be initialized
  - Pseudo instruction ‘ADR’ computes the **offset**
    - Assembler selects the most appropriate ARM instruction (ADD/SUB)
  - Copy data from TABLE1 to TABLE2

```
COPY ADR r1, TABLE1    ; r1 points to TABLE1. Label of COPY
     ADR r2, TABLE2    ; r2 points to TABLE2
...
TABLE1                 ; Source of data
     ...
TABLE2                 : Destination of data
     ...
```

Data Transfer Instructions (III)

- **Using single register load and store instructions**
  - 32-bit Load/store address should be aligned on a 4-byte boundary
  - Modify register ready for the next transfer

```
COPY ADR r1, TABLE1     ; r1 points to TABLE1. Label of COPY
     ADR r2, TABLE2     ; r2 points to TABLE2
LOOP LDR r0, [r1]       ; Load first value
     STR r0, [r2]       ; Store first value
     ADD r1, r1, #4     ; Step r1 on 1 word
     ADD r2, r2, #4     ; Step r2 on 1 word
                         ; If more go back to LOOP
     ...
TABLE1                  ; Source of data
      ...
TABLE2                  : Destination of data
      ...
```

Data Transfer Instructions (IV)

- **Base plus offset addressing**
  - **Pre-indexed** addressing (up to 4K bytes add/sub)
    - `LDR r0, [r1,#4]          ; r0 := mem[r1+4]`
  - **Auto-indexing**
    - `LDR r0, [r1,#4]!         ; r0 := mem[r1+4]. '!': auto-indexing`
      `                         ; r1 := r1 + 4`
  - **Post-indexed** addressing
    - `LDR r0, [r1], #4         ; r0 := mem[r1]`
      `                         ; r1 := r1 + 4`
  - Copy program

```
        ...
LOOP LDR r0, [r1], #4   ; Load first value & post-indexing
        STR r0, [r2], #4   ; Store first value & post-indexing
        ...
```

  - Byte load
    - `LDRB r0, [r1]            ; r0 := mem_8[r1]`

Data Transfer Instructions (V)

- **Multiple register data transfer**
  - Any subset (or all) of the 16 registers to be transferred
  - More restricted addressing modes
  - Transfer list in {}

    -
      ```
      LDMIA r1, {r0,r2,r5}     ; r0 := mem[r1]
                               ; r2 := mem[r1+4]
                               ; r5 := mem[r1+8]
      ```
  - The lowest register is transferred to/from the lowest address
  - Including r15 (PC) in the list will cause a change in the control flow!

Data Transfer Instructions (VI)

- **Stack addressing**
  - Stack
    - Last-in-first-out store which supports simple dynamic memory allocation: Address not known at compile time
    - Ascending stack: grows up
    - Descending stack: grows down
  - Stack pointer
    - Holds the address of the current top of the stack
    - Full stack: pointing to the last valid data pushed
    - Empty stack: pointing to the vacant slot for the next data
  - ARM support
    - Full ascending
    - Empty ascending
    - Full descending
    - Empty descending

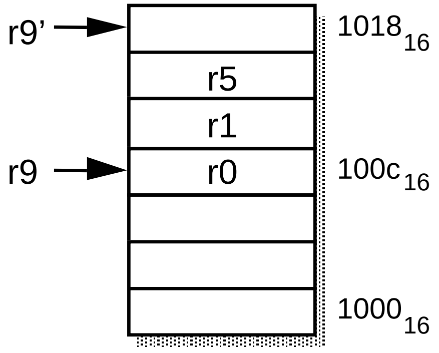
Data Transfer Instructions (VII)

- **Block copy addressing**
  - The mapping between the stack and block copy views of the load and store multiple registers

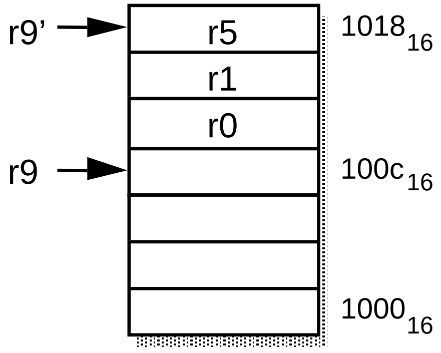
		Ascending		Descending	
		Full	Empty	Full	Empty
Increment	Before	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMFD	
Decrement	Before		LDMDB LDMEA	STMDB STMFD	
	After	LDMDA LDMFA			STMDA STMED

Data Transfer Instructions (VIII)

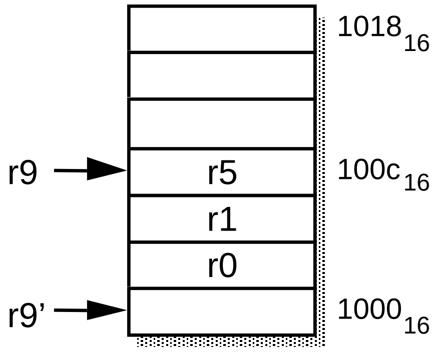
- **Multiple register transfer addressing modes**



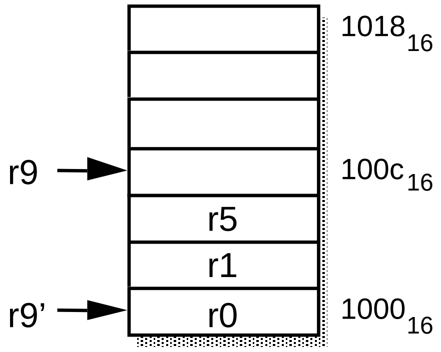
`STMIA r9!, {r0,r1,r5}`



`STMIB r9!, {r0,r1,r5}`



`STMDA r9!, {r0,r1,r5}`



`STMDB r9!, {r0,r1,r5}`

Data Transfer Instructions (IX)

- **Block copy addressing**
  - Block copy instructions
    - STMFD r13!, {r2-r9} ; Save regs onto stack. Full descending.
    - LDMIA r0!, {r2-r9} ;Block load
    - STMIA r1, {r2-r9} ; Block store
    - LDMFD r13!,{r2-r9} ; Restore from stack
  - Efficient way to save and restore processor state and to move blocks of data
  - Operate up to four times faster than single register load/store
  - Data organization in memory in order to maximize the potential for using multiple register data transfer
  - Not pure 'RISC': multiple clock cycles
  - Complex to implement.

4. Control Flow Instructions

- **Branch instructions**
  - Switch program execution
    - B LABEL
    - ...
    - LABEL ...
  - LABEL after or before B instruction

- **Conditional branches**
  - Decision whether or not to branch
  - Control loop exit

```
        MOV r0, #0          ; Initialize counter
LOOP    ...
        ADD r0, r0, #1      ; Increment loop counter
        CMP r0, #10         ; Compare with limit
        BNE LOOP            ; Branch (Repeat) if not equal
        ...                 ; Else fall through
```

Control Flow Instructions (II)

- **Branch instructions**

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Control Flow Instructions (III)

- **Conditional execution**
  - Conditional execution applies not only to branches but to all ARM instructions
  - Example

```
          CMP r0, #5
          BEQ BYPASS        ; if (r0 != 5) {
          ADD r1, r1, r0    ;   r1 := r1 + r0 – r2
          SUB r1, r1, r2    ; }
BYPASS ...
```

  - May be replaced by

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

  - When the conditional sequence is three instructions or fewer
  - Cunning use of conditionals

```
          CMP r0, r1        ; if ((a==b) && (c==d) e++;
          CMPEQ r2, r3
          ADDEQ r4, r4, #1
```

Control Flow Instructions (IV)

- **Branch and link instructions**
  - Functionality for subroutine call & return
    - `        BL SUBR        ; Branch to SUBR`
    - `        ...                  ; Return to here`
    - `SUBR    ...            ; Subroutine entry point`
    - `        MOV pc, r14    ; Return. r14=link register.`
  - *Nested subroutine call: Save r14 and work registers in the stack*
    - `        BL SUB1        ; Branch to SUBR`
    - `        ...                  ; Return to here`
    - `SUB1    STMFD r13!, {r0-r2,r14}   ; Save work & link regs`
    - `        BL SUB2`
    - `        ...`
    - `SUB2    ...`
  - *A subroutine that does not call another subroutine (a **leaf** subroutine) need not save r14 since it will not overwritten.*

Control Flow Instructions (V)

- **Subroutine return instructions**
  - Simplest case
    - SUB2 ...
    - MOV pc, r14 ; Copy r14 (link reg) to r15 (pc) to return
  - Any of data processing instructions can be used to compute a return address
  - When the return address has been pushed onto a stack, it can be restored with any saved working registers using LDM
    - SUB1 STMFD r13!, {r0-r2,r14} ; Save work & link regs
    - BL SUB2
    - ...
    - SUB2 ...
    - LDMFD r13!, {r0-r2, pc} ; Restore work regs & return
    - ; Saved r14 restored to r15 (pc)

Control Flow Instructions (VI)

- **Supervisor calls**
  - When a program requires input or output
  - Operates at a privileged level
  - In many systems the user cannot access hardware facilities directly
  - SWI (Software interrupt) instruction
    - Supervisor call
  - Send a character in bottom r0
    - `SWI SWI_WriteC        ; Output r0[7:0]`
  - Returns control to the monitor program
    - `SWI SWI_Exit          ; Return to monitor`

Control Flow Instructions (VI)

- **Jump tables**
  - When to call one of a set of subroutines
  - Switch statement in C
  - Example

```
        BL      JUMPTAB
        ...
JUMPTAB CMP r0, #0
        BEQ     SUB0
        CMP     r0, #1
        BEQ     SUB1
        CMP     r0, #2
        BEQ     SUB2
```

  - More efficient way

```
        BL      JUMPTAB
        ...
JUMPTAB ADR     r1, SUBTAB ; r1 := SUBTAB
        CMP     r0, #SUBMAX ; Check for
                    overrun
        LDRLS   pc, [r1, r0, LSR #2] ; if OK,
                    table jump
        B       ERROR  ; Else signal error
SUBTAB          DCD     SUB0
        DCD     SUB1
        DCD     SUB2
        ...
```

5. Writing Simple Assembly Language Programs

- **Programming practice**
  - Understand the problem: what to do, input, and output
  - Have a clear idea of your algorithm (top-down)
  - Coding & debugging (bottom-up)

- **Software development toolkit**
  - Text editor to type the program into
  - Assembler to turn the program into ARM binary code
  - ARM system emulator to execute the binary on.
    - Some text output capability.
  - Debugger to see what is happening inside your program.

Writing Simple Assembly Programs (II)

- **Hello world program**
  - Print ‘Hello world’ on the display

```
        AREA HelloW, CODE, READONLY ; Declare code area
SWI_WriteC   EQU     &0             ; Output character in r0
SWI_Exit     EQU     &11            ; Finish program System call
        ENTRY                       ; Code entry point System call
START ADR    r1, TEXT               ; r1 -> “Hello World”. Pseudo instr
LOOP  LDRB   r0, [r1], #1           ; Get the next byte. Auto indexed
        CMP    r0, #0               ; Check for text end
        SWINE  SWI_WriteC           ; If not end, print.
                                    ;  Conditional syscall
        BNE    LOOP                 ; .. And loop back
        SWI    SWI_Exit             ; End of execution
TEXT   = “Hello World”,&0xa,&0xd,0  ; Text string:  null terminated
        END                         ; End of program source
```

Writing Simple Assembly Programs (III)

- **Test block copy program**
  - Block copy a text string & output

```
        AREA    BlkCpy, CODE, READONLY
SWI_WriteC      EQU     &0      ; Output char in r0
SWI_Exit        EQU     &11     ; Finish program
        ENTRY                   ; Code entry point
        ADR     r1, TABLE1      ; r1 -> TABLE1
        ADR     r2, TABLE2      ; r2 -> TABLE2
        ADR     r3, T1END       ; r3 -> T1END
LOOP1   LDR     r0, [r1], #4    ; Get TABLE1 1st word
        STR     r0, [r2], #4    ; Copy into TABLE2
        CMP     r1, r3          ; Finished?
        BLT     LOOP1           ; If not, do more. BNE may fail!
        ADR     r1, TABLE2      ; r1 -> TABLE1
                        ; To be continued
```

Writing Simple Assembly Programs (IV)

- ***Test block copy program (cont'd)***

```
LOOP2 LDRB    r0, [r1], #1        ; Get next byte
      CMP     r0, #0              ; Check for text end
      SWINE   SWI_WriteC          ; Print character
      BNE     LOOP2               ; Loop back
      SWI     SWI_Exit            ; Finish
 ; String data area
      ALIGN                       ; Ensure word alignment
TABLE1        = "This is the right string!", &0a, &0d, 0
T1END
      ALIGN                       ; Ensure word alignment
TABLE2        = "This is the wrong string!", &0a, &0d, 0
      END
```

Writing Simple Assembly Programs (V)

- **Program design**
  - Pile of simple programs != complex programs
  - Serious programming
    - Should not start with coding, but with careful design

    - 1. Understand the requirements
    - 2. Requirements should be translated into an unambiguous specification
    - 3. Define a program structure & data structure
    - 4. Devise suitable algorithms in **pseudo-code**
      - A program-like notation which does not follow the syntax of a particular programming language but which makes the meaning clear
    - 5. Begin coding
      - Individual modules should be coded, tested thoroughly, and documented.

  - *It may be necessary to develop small software components in assembly language to get the best performance for a critical application.*