

## Week 2. ARM Assembly Programming

### 18-342: Fundamentals of Embedded Systems

**Rajeev Gandhi**

INI & ECE  
Carnegie Mellon University





















**Carnegie Mellon**




### Overview of this Week

- ARM Architecture
  - Program Status Register
  - Exception handling
- Overview of ARM's instruction set
  - Data processing instructions
  - Use of the barrel shifter
- Conditional execution
- **Announcement:**
  - Quiz next Tuesday
  - Open books, open notes (no sharing of notes)
  - Bring a calculator (if you cannot do binary/hex arithmetic)
  - No laptops, cell phones or any other device that can be used to communicate with others
  - Useful tip: **understand** and print the 4-page ARM instruction summary available at Blackboard ARM programming reference card

## ARM's Register Organization

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
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	 R13-fiq	 R13-svc	 R13-abt	 R13-irq	 R13-und
R14	 R14-fiq	 R14-svc	 R14-abt	 R14-irq	 R14-und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	 SPSR-fiq	 SPSR-svc	 SPSR-abt	 SPSR-irq	 SPSR-und

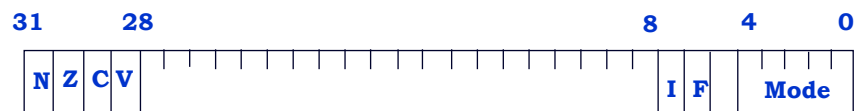
 = banked register

SPSR = State Program Status Register

3

## Current Program Status Register

- ◆ Current Program Status Register (`cpsr`) is a dedicated register
- ◆ Holds information about the most recently performed ALU operation
- ◆ Controls the enabling and disabling of interrupts (both IRQ and FIQ)
- ◆ Sets the processor operating mode
- ◆ Sets the processor state



4

## Current Program Status Register

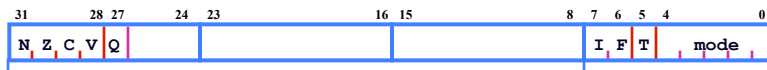


- ◆ cpsr has two important pieces of information
  - Flags: contains the condition flags
  - Control: contains the processor mode, state and interrupt mask bits
- ◆ All fields of the cpsr can be read/written in privileged modes
- ◆ Only the flag field of cpsr can be written in User mode, all fields can be read in User mode
- ◆ Notation: we will use cpsr\_f to refer to the flag fields of cpsr, cpsr\_c to refer to control fields of cpsr, cpsr\_cf to refer to both the flag and control fields of the cpsr ...

M[4:0]	Mode
10000	User
10001	FIQ
10010	IRQ
10011	SVC
10111	Abort
11011	Undef
11111	System

5

## Current Program Status Register (contd.)



- Condition code flags
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed
- Sticky Overflow flag - Q flag
  - Architecture 5TE/J only
  - Indicates if saturation has occurred
- Interrupt Disable bits
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.
- T Bit
  - Architecture xT only
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- Mode bits
  - Specify the processor mode

Remember – when an exception occurs the cpsr gets copied to the corresponding spsr\_<mode>

6

## Thumb

- ◆ Thumb is a 16-bit instruction set
  - Optimized for code density from C code (~65% of ARM code size)
  - Improved performance from narrow memory
  - Subset of the functionality of the ARM instruction set
- ◆ Core has additional execution state - Thumb
  - Switch between ARM and Thumb using **BX** instruction

**ADDS r2, r2, #1**

32-bit ARM Instruction



**ADD r2, #1**

16-bit Thumb Instruction

For most instructions generated by compiler:

- Conditional execution is not used
- Source and destination registers identical
- Only Low registers used
- Constants are of limited size
- Inline barrel shifter not used

7

## Exceptions

- ◆ Kind of interrupt that might be used to signal (and therefore handle) a faulty condition
- ◆ What kinds of exceptional conditions can occur?
  - Powering the system off completely
  - Using an instruction that is not defined
  - Accessing a memory location that is not aligned
  - Accessing a memory location that is not permitted
- ◆ Sometimes an external interrupt can also be defined as an exception

8

## Exceptions vs. Interrupts

- ◆ The terms *exception* and *interrupt* are often confused
- ◆ *Exception* usually refers to an internal CPU event such as
  - Floating point overflow
  - MMU fault (e.g., page fault)
  - Trap (SWI)
- ◆ *Interrupt* usually refers to an external I/O event such as
  - I/O device request
  - Reset
- ◆ In the ARM architecture manuals, the two terms are mixed together
- ◆ In the context of the ARM processor, the two terms are interchangeable

9

## Exceptions

Any condition that halts the normal sequential execution of program instructions

Exception	Mode	Description
Reset	Supervisor	Occurs when the processor's reset button is asserted. This exception is only expected to occur for signaling power up or for resetting the processor. A soft reset can be achieved by branching to reset vector 0x00000000
Undefined Instruction	Undef	Occurs if neither the processor, nor any of the coprocessors, recognize the currently executing instruction
Software Interrupt	Supervisor	This is a user-defined synchronous interrupt. It allows a program running in the User mode to request privileged operations (for example an RTOS function) that run in Supervisor mode
Prefetch Abort	Abort	Occurs when a processor attempts to execute an instruction that was not fetched, because the address was illegal
Data Abort	Abort	Occurs when a data transfer instruction attempts to load or store data at an illegal address
IRQ	IRQ	Occurs when the processor's external interrupt request pin is asserted and the I bit in the <code>cpsr</code> is clear
FIQ	FIQ	Occurs when the processor's external fast interrupt request pin is asserted and the I bit in the <code>cpsr</code> is clear

10

## Exception Handling

- ◆ **Exception Handler**
  - Most exceptions have an associated software exception handler that executes when that particular exception occurs
- ◆ **Where is this exception handler located?**
- ◆ **Vector table**
  - Reserved area of 32 bytes at the end of the memory map (starting at address 0x0)
  - One word of space for each exception type
  - Contains a Branch or Load PC instruction for the exception handler
- ◆ **Exception modes and registers**
  - Handling exceptions changes program from user to non-user mode
  - Each exception handler has access to its own set of registers
    - Its own `r13` (stack pointer)
    - Its own `r14` (link register)
    - Its own `spsr` (Saved Program Status Register)
  - Exception handlers must save (restore) other register on entry (exit)

11

## Exception Handling

- ◆ **When an exception occurs, the ARM:**
  - Copies `cpsr` into `spsr_<mode>`
  - Sets appropriate `cpsr` bits
    - Change to ARM state
    - Change to exception mode
    - Disable interrupts (if appropriate)
  - Stores the return address in `lr_<mode>`
  - Sets `pc` to vector address
- ◆ **To return, exception handler needs to:**
  - Restore `cpsr` from `spsr_<mode>`
  - Restore `pc` from `lr_<mode>`

	⋮
0x1C	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0C	Prefetch Abort
0x08	Software Interrupt
0x04	Undefined Instruction
0x00	Reset

**Vector Table**

12

## Simultaneously Occurring Exceptions?

Vector address	Exception type	Exception mode	Priority (1=high, 6=low)
0x0	Reset	Supervisor (SVC)	1
0x4	Undefined Instruction	Undef	6
0x8	Software Interrupt (SWI)	Supervisor (SVC)	6
0xC	Prefetch Abort	Abort	5
0x10	Data Abort	Abort	2
0x14	<i>Reserved</i>	<i>Not applicable</i>	<i>Not applicable</i>
0x18	Interrupt (IRQ)	Interrupt (IRQ)	4
0x1C	Fast Interrupt (FIQ)	Fast Interrupt (FIQ)	3

13

## Exceptions Are Not Always Bad

- ◆ ARM exceptions can be used to provide functionality that would not have been otherwise possible
  - Memory or Data Abort exception can be used to detect and deal with page-faults in an ARM system that uses Virtual Memory
- ◆ Undefined exception can be used to provide software emulation of coprocessor when the coprocessor is not physically present or for special purpose instruction set extension
  - When the processor encounters an instruction which is not part of the ARM instruction set, the processor changes to Undefined mode and executes the Undefined Instruction exception handler
  - In the exception handler, the coprocessor functionality can be provided in software (or the functionality provided by the enhanced instructions can be provided in software)

14

## The ARM Assembly Language

- ◆ ARM instructions can be broadly classified as
  - **Data Processing Instructions**: manipulate data within the registers
  - **Branch Instructions**: changes the flow of instructions or call a subroutine
  - **Load-Store Instructions**: transfer data between registers and memory
  - **Software Interrupt Instruction**: causes a software interrupt
  - **Program Status Instructions**: read/write the processor status registers
- ◆ All instructions can access `r0–r14` directly
- ◆ Most instructions also allow use of the `pc`
- ◆ Specific instructions to allow access to `cpsr` and `spsr`

15

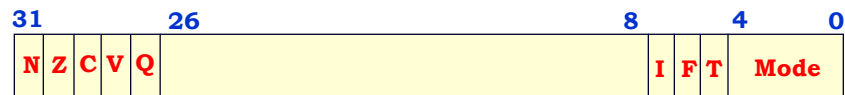
## Reminder: The Program Counter (`r15`)

- ◆ When the processor is executing in ARM state:
  - All instructions are 32 bits in length
  - 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).
- ◆ `r15` is the program counter
- ◆ `r14` is used as the subroutine link register (`lr`) and stores the return address when Branch with Link (BL) operations are performed
- ◆ Thus to return from a linked branch
  - `MOV r15, r14`or
  - `MOV pc, lr`

16



## Reminder: `cpsr`



- ◆ Will represent this as `nzcvqift_mode`
- ◆ Capital letter will indicate that a certain bit has been set
- ◆ Examples
  - `nzcvqift_USER`: FIQs are masked and the processor is executing in user mode
  - `nzCvqift_SVC`: Carry flag is set and the processor is executing in supervisor mode

17

## Data Processing Instructions

- ◆ Manipulate data within registers
  - Move operations
  - Arithmetic operations
  - Logical operations
  - Comparison operations
  - Multiply operations
  
- ◆ Appending the S suffix for an instruction, e.g., ADDS
  - Signifies that the instruction's execution will update the flags in the `cpsr`

18

## Typical ARM Data Processing Instruction



- ◆ **Operation** – Specifies the instruction to be performed
- ◆ Almost all ARM instructions can be conditionally executed
- ◆ **Cond** – specify the optional conditional flags which have to be set under which to execute the instruction
- ◆ **S bit** – Signifies that the instruction updates the conditional flags
- ◆ **Rd** – Specifies the destination register
- ◆ **Rn** – Specifies the first source operand register
- ◆ **ShifterOperand2** – Specifies the second source operand
  - Could be a register, immediate value, or a shifted register/immediate value
- ◆ Some data processing instructions may not specify the destination register or the source register

19

## Data Processing Instructions

- ◆ Consist of :
  - Arithmetic:     **ADD   ADC   SUB   SBC   RSB   RSC**
  - Logical:       **AND   ORR   EOR   BIC**
  - Comparisons:   **CMP   CMN   TST   TEQ**
  - Data movement: **MOV   MVN**
- ◆ These instructions only work on registers, NOT memory

20

## Move Instructions

- ◆ MOV moves a 32-bit value into a register
- ◆ MVN moves the NOT of the 32-bit value into a register

PRE    r5 = 5  
       r7 = 8

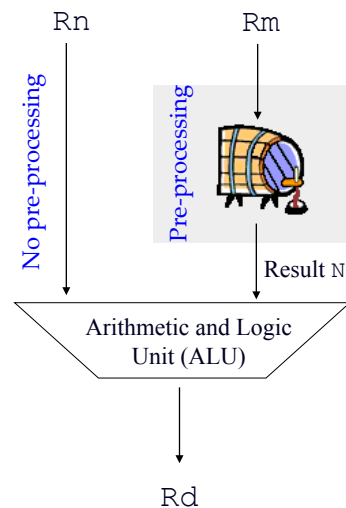
MOV r7, r5

POST  r5 =  
       r7 =

21

## The ARM Barrel Shifter

- ◆ Data processing instructions are processed within the ALU
- ◆ ARM can shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before the value enters the ALU
- ◆ Can achieve fast multiplies or division by a power of 2
- ◆ Data-processing instructions that do not use the barrel shifter
  - MUL (multiply)
  - CLZ (count leading zeros)
  - QADD (signed saturated 32-bit add)



22

## Using the Barrel Shifter

- ◆ LSL shifts bits to the left, and is similar to the C-language operator <<

```
PRE   r5 = 5
      r7 = 8
```

```
MOV r7, r5, LSL #2 ← r7 = r5 * 4 = (r5 << 2)
```

```
POST  r5 =
      r7 =
```

23

## Updating the Condition Flags

- ◆ The S suffix indicates that the `cpsr` should be updated

```
PRE   cpsr = nzcvtFt_USER
      r0 = 0x00000000
      r1 = 0x80000004
```

```
MOVS r0, r1, LSL #1
```

```
POST  cpsr =
      r0 =
      r1 =
```

Hint: 32-bit version of 0x80000004 is  
1000 0000 0000 0000 0000 0000 0000 0100

24

## Arithmetic Instructions

- ◆ Addition and subtraction of 32-bit signed and unsigned values

### ◆ Subtraction

```
PRE   r0 = 0x00000000
      r1 = 0x00000002
      r2 = 0x00000001
```

```
SUB r0, r1, r2
```

```
POST  r0 =
      r1 =
      r2 =
```

### ◆ Addition

```
PRE   r0 = 0x00000000
      r1 = 0x00000005
```

```
ADD r0, r1, r1, LSL #1
```

```
POST  r0 =
      r1 =
```

25

## Logical Instructions

- ◆ Bitwise logical operations on two source registers
- ◆ AND, ORR, EOR, BIC

Every binary 1 in r2  
clears a corresponding  
bit location in r1

### ◆ Logical OR

```
PRE   r0 = 0x00000000
      r1 = 0x02040608
      r2 = 0x10305070
```

```
ORR r0, r1, r2
```

```
POST  r0 =
      r1 =
      r2 =
```

### ◆ Logical bit clear (BIC)

```
PRE   r1 = 0b1111
      r2 = 0b0101
```

```
BIC r0, r1, r2
```

```
POST  r0 =
      r1 =
      r2 =
```

26

## Comparison Instructions

- ◆ Compare or test a register with a 32-bit value
  - CMN, CMP, TEQ, TST
- ◆ Outcome: Registers under comparison are not affected; cpsr updated
  - Example: CMP x, y sets cpsr flags based on results of x-y (subtract)
  - Example: TST x, y sets cpsr flags based on results of x&y (logical AND)
- ◆ Do not need the S suffix

```
PRE    cpsr = nzcvcifT_USER
        r0 = 4
        r9 = 4
```

```
CMP r0, r9
```

```
POST   cpsr =
        r0 =
        r9 =
```

27

## Multiply Instructions

- ◆ Multiple a pair of registers and optionally add (accumulate) the value stored in another register
  - MUL Rd, Rm, Rs  $\Rightarrow$  Rd = Rm\*Rs
  - MLA Rd, Rm, Rs, Rn  $\Rightarrow$  Rd = Rm\*Rs + Rn
- ◆ Special instructions called long multiplies accumulate onto a pair of registers representing a 64-bit value
  - SMLAL, SMULL, UMLAL, UMUL

```
PRE    r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000002
```

```
MUL r0, r1, r2
```

```
POST   r0 =
        r1 =
        r2 =
```

28

## ARM Instruction Set Encoding

- ◆ Remember we said that all ARM instructions were 32 bits long?
- ◆ All ARM instructions encoded
  - Contains condition code
  - Information about whether the processor changes mode
  - Register list, a bit field where a bit is set if the corresponding register is used
  - Writeback addressing mode employed
  - And much more
- ◆ What constraints does this impose on operand lengths and on what is possible in each operation?

29

## Conditional Execution

- ◆ Most ARM instructions are conditionally executed
  - Instruction executes only if the condition-code flags satisfy a given test
- ◆ Increases performance
  - Reduces the number of branches, which reduces the number of pipeline flushes
- ◆ Improves code density
- ◆ Two-letter mnemonic appended to the instruction mnemonic
  - Examples: **BEQ**, **ADDNE**, **SUBLT**
- ◆ Example usage
  - An add operation takes the form
    - `ADD r0,r1,r2` ; `r0 = r1 + r2`
  - To execute this only if the zero flag is set:
    - `ADDEQ r0,r1,r2` ; If zero flag set then...
    - ; ... `r0 = r1 + r2`

30

## ARM Instruction Set Encoding

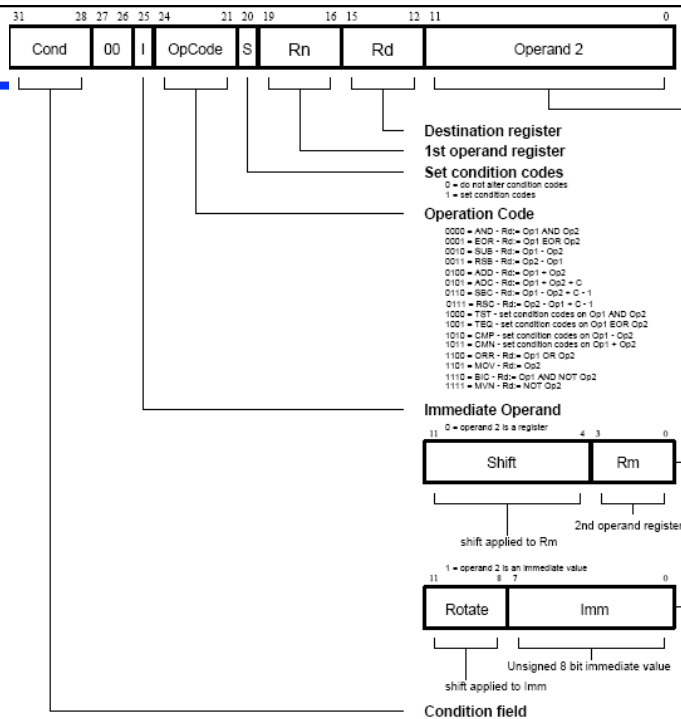
3 3 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0  
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

Cond	0	0	I	Opcode	S	Rn	Rd	Operand 2										<i>Data Processing / PSR Transfer</i>																										
Cond	0	0	0	0	0	0	A S	Rd	Rn	Rs	1	0	0	1	Rm	<i>Multiply</i>																												
Cond	0	0	0	0	1	0	U A S	RdHi	RdLo	Rn	1	0	0	1	Rm	<i>Multiply Long</i>																												
Cond	0	0	0	1	0	B	0	Rn	Rd	0	0	0	1	0	0	1	Rm	<i>Single Data Swap</i>																										
Cond	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	Rn	<i>Branch and Exchange</i>																									
Cond	0	0	0	P	U	0	W L	Rn	Rd	0	0	0	1	S	H	1	Rm	<i>Halfword Data Transfer: register offset</i>																										
Cond	0	0	0	P	U	1	W L	Rn	Rd	Offset		1	S	H	1	Offset	<i>Halfword Data Transfer: immediate offset</i>																											
Cond	0	1	1	P	U	B	W L	Rn	Rd	Offset										<i>Single Data Transfer</i>																								
Cond	0	1	1													1	<i>Undefined</i>																											
Cond	1	0	0	P	U	S	W L	Rn	Register List												<i>Block Data Transfer</i>																							
Cond	1	0	1	L	Offset												<i>Branch</i>																											
Cond	1	1	0	P	U	N	W L	Rn	CRd	CP#	Offset										<i>Coprocessor Data Transfer</i>																							
Cond	1	1	1	0	CP	Opc		CRn	CRd	CP#	CP	0			CRm	<i>Coprocessor Data Operation</i>																												
Cond	1	1	1	0	CP	Opc	L	CRn	Rd	CP#	CP	1			CRm	<i>Coprocessor Register Transfer</i>																												
Cond	1	1	1	1	Ignored by processor												<i>Software Interrupt</i>																											

3 3 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0  
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

31

## Data Processing Instructions

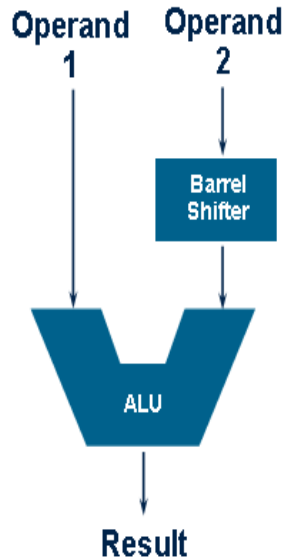


32



## More on Barrel Shifter

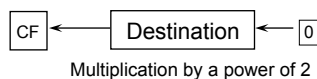
- ◆ Shift value can be specified by either an immediate constant or as a register
- ◆ Instruction do not take any longer to complete
  - unless the shift is specified by a register
- ◆ Operand 2 can be
  - Register
    - Shift value can be either be:
      - 5 bit unsigned integer
      - Specified in bottom byte of another register
  - Immediate constant:
    - 8 bit constant rotated through an even number of positions



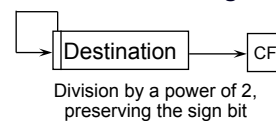
33

## The Barrel Shifter

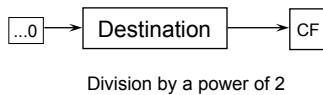
### LSL : Logical Left Shift



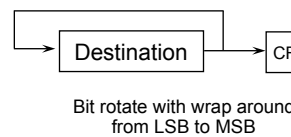
### ASR: Arithmetic Right Shift



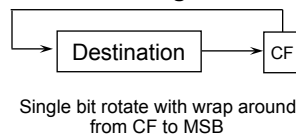
### LSR : Logical Shift Right



### ROR: Rotate Right



### RRX: Rotate Right Extended



34

## Use of Barrel Shifter

- ◆ Any data processing instruction can use the barrel shifter to process the second operand (shifter operand)
- ◆ Shifter operand could be
  1. An 8-bit immediate constant rotated right by 0, 2, 4, 6, ... 30. In the instruction, the rotate amount is specified to be an even number 0,2,4...30

In this case the shifter operand is of the form `#constant{, #rotation-amount}`

Examples:

<code>MOV r0, #32;</code>	<code>r0 ↵ 32</code>
<code>MOV r0, #32, #2;</code>	<code>r0 ↵ 8</code>
<code>MOV r1, #0xFF, #4;</code>	<code>r1 ↵ 0xF00000F</code>

35

## Use of Barrel Shifter

2. A register shifted (LSL, LSR, ASR or ROR) by a constant (from 0 to 31) where the amount of shift is specified by an immediate value

In this case the shifter operand is of the form `register {, shift shift-amount}`

Examples:

<code>SUB r0, r1, r2, LSL #2;</code>	<code>r0 ↵ r1-4*r2</code>
<code>RSB r0, r1, r2, LSL #2;</code>	<code>r0 ↵ 4*r2 - r1</code>

All multiplications by a constant which is a power of two +/- 1 can be efficiently implemented using the barrel shifter

Examples:

<code>r0 = r1 * 8</code>	<code>MOV r0, r1, LSL #3</code>
<code>r0 = r1 * 9</code>	
<code>= r1+(r1*8)</code>	<code>ADD r0, r1, r1, LSL#3</code>

36

## Use of Barrel Shifter

3. A register shifted (LSL, LSR, ASR or ROR) by a constant where the amount of shift is specified by another register

In this case the shifter operand is of the form `register {, shift register}`

Examples:

```
ADD r0, r1, r2 ROR r5;  
SUB r3, r2, r1, LSL r0;
```

4. A register rotate right extended

In this case the shifter operand is of the form `register, RRX`

Examples:

```
MOV r3, r2, RRX;
```

37

## Branch Instructions

- ◆ To change the flow of execution or to call a routine
- ◆ Supports subroutine calls, *if-then-else* structures, loops
- ◆ Change of execution forces the `pc` to point to a new address
- ◆ Four different branch instructions on the ARM
  - `B{<cond>} label`
  - `BL{<cond>} label`
  - `BX{<cond>} Rm`
  - `BLX{<cond>} label | Rm`

38

## Condition Mnemonics

Suffix/Mnemonic	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N!=V
AL	Always	

39

## Value of Conditional Execution

- ◆ This improves code density *and* performance by reducing the number of forward branch instructions

```

if (x != 0)
    a = b+c;
else
    a = b-c;

CMP    r3,#0
BEQ    skip
ADD    r0,r1,r2
B      afterskip
skip
SUB    r0, r1, r2
afterskip

```

```

CMP    r3,#0
ADDNE  r0,r1,r2
SUBEQ  r0,r1,r2

```

- ◆ Can also be used to optimize a countdown loop (where the loop variable decrements from a positive number to zero)

```

loop
    . . .
    SUB r1,r1,#1
    CMP r1, #0
    BNE loop

```

```

loop
    . . .
    SUBS r1,r1,#1
    BNE loop

```

40

## Examples of Conditional Execution

- ◆ Use a sequence of several conditional instructions

```
if (a==0) x=1;  
    CMP     r0,#0  
    MOVEQ   r1,#1
```

- ◆ Set the flags, then use various condition codes

```
if (a==0) x=0;  
if (a>0)  x=1;  
    CMP     r0,#0  
    MOVEQ   r1,#0  
    MOVGT   r1,#1
```

- ◆ Use conditional compare instructions

```
if (a==4 || a==10) x=0;  
    CMP     r0,#4  
    CMPNE   r0,#10  
    MOVEQ   r1,#0
```

41