ARM Assembly Programming

Computer Organization and Assembly Languages

Yung-Yu Chuang

2007/11/19

with slides by Peng-Sheng Chen

Introduction

- The ARM processor is very easy to program at the assembly level. (It is a RISC)
- We will learn ARM assembly programming at the user level and run it on a GBA emulator.

ARM programmer model

- The state of an ARM system is determined by the content of visible registers and memory.
- A user-mode program can see 15 32-bit general-purpose registers (R0-R14), program counter (PC) and CPSR.
- Instruction set defines the operations that can change the state.

Memory system

- Memory is a linear array of bytes addressed from 0 to 2^{32}-1
- Word, half-word, byte
- Little-endian

0x00000000	00
0x00000001	10
0x00000002	20
0x00000003	30
0x00000004	FF
0x00000005	FF
0x00000006	FF
0xFFFFFFFD	00
0xFFFFFFFE	00
0xFFFFFFFF	00

Byte ordering

- Big Endian
  - Least significant byte has highest address

  Word address 0x00000000

  Value: 00102030
- Little Endian
  - Least significant byte has lowest address

  Word address 0x00000000

  Value: 30201000

0x00000000	00
0x00000001	10
0x00000002	20
0x00000003	30
0x00000004	FF
0x00000005	FF
0x00000006	FF
0xFFFFFFFD	00
0xFFFFFFFE	00
0xFFFFFFFF	00

ARM programmer model

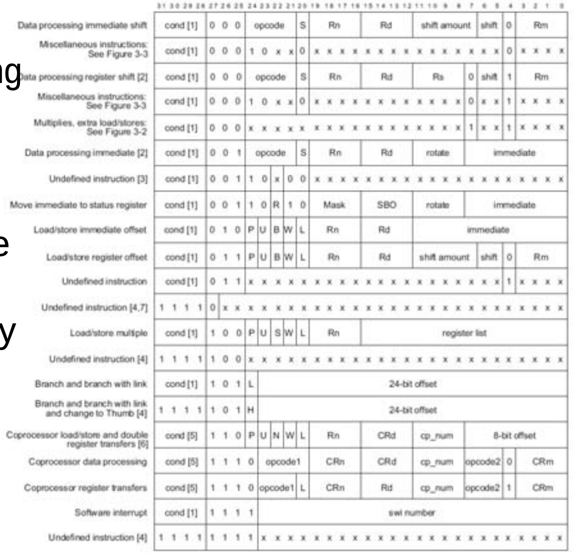
R0	R1	R2	R3
R4	R5	R6	R7
R8	R9	R10	R11
R12	R13	R14	PC

31	30	29	28	27	26		8	7	6	5	4	3	2	1	0
N	Z	C	V	Q				I	F	T	M 4	M 3	M 2	M 1	M 0

0x00000000	00
0x00000001	10
0x00000002	20
0x00000003	30
0x00000004	FF
0x00000005	FF
0x00000006	FF
0xFFFFFFFD	00
0xFFFFFFFE	00
0xFFFFFFFF	00

Instruction set

ARM instructions are all 32-bit long (except for Thumb mode). There are 2^{32} possible machine instructions. Fortunately, they are structured.



	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data processing immediate shift	cond [1]				0	0	0	opcode				S	Rn				Rd				shift amount					shift		0	Rm			
Miscellaneous instructions: See Figure 3-3	cond [1]				0	0	0	1	0	x	x	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	x	x	x	x
Data processing register shift [2]	cond [1]				0	0	0	opcode				S	Rn				Rd				Rs				0	shift		1	Rm			
Miscellaneous instructions: See Figure 3-3	cond [1]				0	0	0	1	0	x	x	0	x	x	x	x	x	x	x	x	x	x	x	x	0	x	x	1	x	x	x	x
Multiplies, extra load/stores: See Figure 3-2	cond [1]				0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	x	x	1	x	x	x	x
Data processing immediate [2]	cond [1]				0	0	1	opcode				S	Rn				Rd				rotate				immediate							
Undefined instruction [3]	cond [1]				0	0	1	1	0	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Move immediate to status register	cond [1]				0	0	1	1	0	R	1	0	Mask				SBO				rotate				immediate							
Load/store immediate offset	cond [1]				0	1	0	P	U	B	W	L	Rn				Rd				immediate											
Load/store register offset	cond [1]				0	1	1	P	U	B	W	L	Rn				Rd				shift amount					shift		0	Rm			
Undefined instruction	cond [1]				0	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	x	x	x	x
Undefined instruction [4,7]	1	1	1	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Load/store multiple	cond [1]				1	0	0	P	U	S	W	L	Rn				register list															
Undefined instruction [4]	1	1	1	1	1	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Branch and branch with link	cond [1]				1	0	1	L	24-bit offset																							
Branch and branch with link and change to Thumb [4]	1	1	1	1	1	0	1	H	24-bit offset																							
Coprocessor load/store and double register transfers [6]	cond [5]				1	1	0	P	U	N	W	L	Rn				CRd				cp_num				8-bit offset							
Coprocessor data processing	cond [5]				1	1	1	0	opcode1				CRn				CRd				cp_num				opcode2			0	CRm			
Coprocessor register transfers	cond [5]				1	1	1	0	opcode1			L	CRn				Rd				cp_num				opcode2			1	CRm			
Software interrupt	cond [1]				1	1	1	1	swi number																							
Undefined instruction [4]	1	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Features of ARM instruction set

- Load-store architecture
- 3-address instructions
- Conditional execution of every instruction
- Possible to load/store multiple register at once
- Possible to combine shift and ALU operations in a single instruction

Instruction set

```
MOV<cc><S>  Rd, <operands>

MOVCS R0, R1 @ if carry is set
             @ then R0:=R1

MOVS  R0, #0 @ R0:=0
             @ Z=1, N=0
             @ C, V unaffected
```

Instruction set

- Data processing (Arithmetic and Logical)
- Data movement
- Flow control

Data processing

- Arithmetic and logic operations
- General rules:
  - All operands are 32-bit, coming from registers or literals.
  - The result, if any, is 32-bit and placed in a register (with the exception for long multiply which produces a 64-bit result)
  - 3-address format

Arithmetic

- `ADD  R0, R1, R2      @ R0 = R1+R2`
- `ADC  R0, R1, R2      @ R0 = R1+R2+C`
- `SUB  R0, R1, R2      @ R0 = R1-R2`
- `SBC  R0, R1, R2      @ R0 = R1-R2+C-1`
- `RSB  R0, R1, R2      @ R0 = R2-R1`
- `RSC  R0, R1, R2      @ R0 = R2-R1+C-1`

Bitwise logic

- `AND  R0, R1, R2   @ R0 = R1 and R2`
- `ORR  R0, R1, R2   @ R0 = R1 or  R2`
- `EOR  R0, R1, R2   @ R0 = R1 xor R2`
- `BIC  R0, R1, R2   @ R0 = R1 and (~R2)`

bit clear: **R2** is a mask identifying which
bits of **R1** will be cleared to zero

```
R1=0x11111111    R2=0x01100101

BIC R0, R1, R2

R0=0x10011010
```

Register movement

- `MOV  R0, R2     @ R0 = R2`
- `MVN  R0, R2     @ R0 = ~R2`

move negated

Comparison

- These instructions do not generate a result, but set condition code bits (N, Z, C, V) in CPSR. Often, a branch operation follows to change the program flow.

compare
- `CMP  R1, R2     @ set cc on R1-R2`

compare negated
- `CMN  R1, R2     @ set cc on R1+R2`

bit test
- `TST  R1, R2     @ set cc on R1 and R2`

test equal
- `TEQ  R1, R2     @ set cc on R1 xor R2`

Addressing modes

- Register operands

```
ADD  R0, R1, R2
```

- Immediate operands

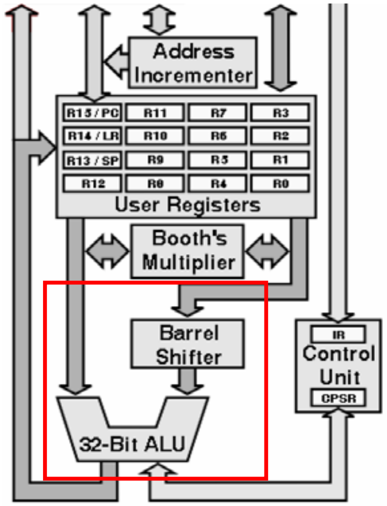
a literal; most can be represented by (0..255)x2^{2n} 0<n<12

```
ADD  R3, R3, #1    @ R3:=R3+1
AND  R8, R7, #0xff @ R8=R7[7:0]
```

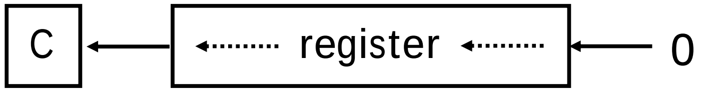
a hexadecimal literal
This is assembler dependent syntax.

Shifted register operands

- One operand to ALU is routed through the Barrel shifter. Thus, the operand can be modified before it is used. Useful for dealing with lists, table and other complex data structure. (similar to the displacement addressing mode in CISC.)

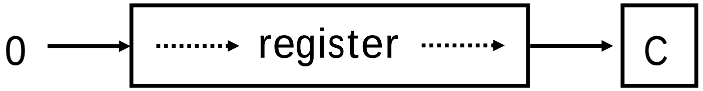


Logical shift left



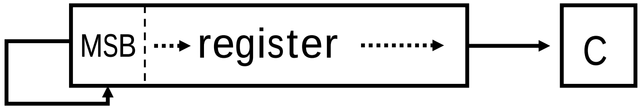
```
MOV   R0, R2, LSL #2 @ R0:=R2<<2
                     @ R2 unchanged
Example: 0…0 0011 0000
Before R2=0x00000030
After  R0=0x000000C0
       R2=0x00000030
```

Logical shift right



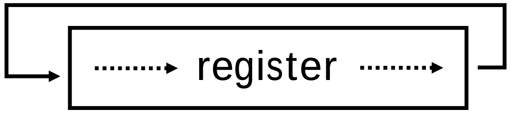
```
MOV   R0, R2, LSR #2 @ R0:=R2>>2
                     @ R2 unchanged
Example: 0…0 0011 0000
Before R2=0x00000030
After  R0=0x0000000C
       R2=0x00000030
```

Arithmetic shift right



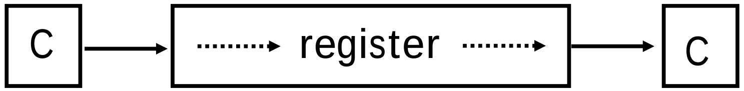
```
MOV   R0, R2, ASR #2 @ R0:=R2>>2
                     @ R2 unchanged
Example: 1010 0…0 0011 0000
Before R2=0xA0000030
After  R0=0xE800000C
       R2=0xA0000030
```

Rotate right



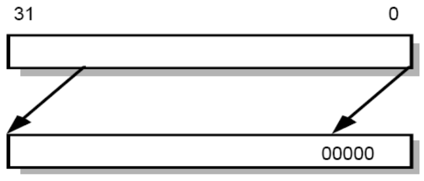
```
MOV  R0, R2, ROR #2 @ R0:=R2 rotate
                    @ R2 unchanged
Example: 0…0 0011 0001
Before R2=0x00000031
After  R0=0x4000000C
       R2=0x00000031
```

Rotate right extended

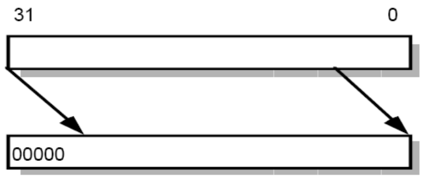


```
MOV  R0, R2, RRX    @ R0:=R2 rotate
                    @ R2 unchanged
Example: 0…0 0011 0001
Before R2=0x00000031, C=1
After  R0=0x80000018, C=1
       R2=0x00000031
```

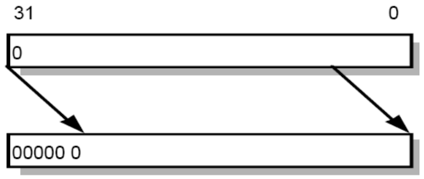
Shifted register operands



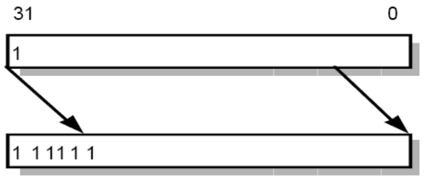
LSL #5



LSR #5

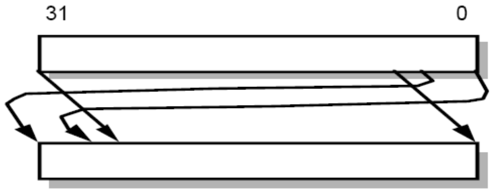


ASR #5 , positive operand

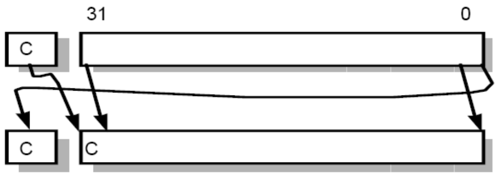


ASR #5 , negative operand

Shifted register operands



ROR #5



RRX

Shifted register operands

- It is possible to use a register to specify the number of bits to be shifted; only the bottom 8 bits of the register are significant.

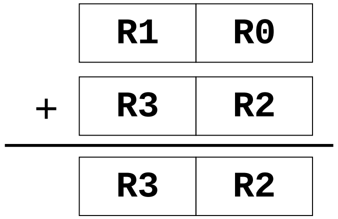
ADD R0, R1, R2, LSL R3 @ R0:=R1+R2*2^{R3}

Setting the condition codes

- Any data processing instruction can set the condition codes if the programmers wish it to

64-bit addition

```
ADDS  R2, R2, R0
ADC   R3, R3, R1
```



Multiplication

- **MUL R0, R1, R2 @ R0 = (R1xR2)$_{[31:0]}$**
- Features:
  - Second operand can't be immediate
  - The result register must be different from the first operand
  - If S bit is set, C flag is meaningless
- See the reference manual (4.1.33)

Multiplication

- Multiply-accumulate

```
MLA  R4, R3, R2, R1  @ R4 = R3xR2+R1
```

- Multiply with a constant can often be more efficiently implemented using shifted register operand

```
MOV  R1, #35
MUL  R2, R0, R1
```

or

```
ADD  R0, R0, R0, LSL #2  @ R0'=5xR0
RSB  R2, R0, R0, LSL #3  @ R2 =7xR0'
```

Data transfer instructions

- Move data between registers and memory
- Three basic forms
  - Single register load/store
  - Multiple register load/store
  - Single register swap: **SWP(B),** atomic instruction for semaphore

Single register load/store

- The data items can be a 8-bitbyte, 16-bit half-word or 32-bit word.

LDR R0, [R1]   @ R0 := mem$_{32}$**[R1]**

STR R0, [R1]   @ mem$_{32}$**[R1] := R0**

LDR, LDRH, LDRB for 32, 16, 8 bits

STR, STRH, STRB for 32, 16, 8 bits

Load an address into a register

- The pseudo instruction **ADR** loads a register with an address

```
table:   .word    10
…
         ADR  R0, table
```

- Assembler transfer pseudo instruction into a sequence of appropriate instructions

```
  sub    r0, pc, #12
```

Addressing modes

- Memory is addressed by a register and an offset.

```
    LDR  R0, [R1]  @ mem[R1]
```

- Three ways to specify offsets:
  - Constant

```
    LDR  R0, [R1, #4]  @ mem[R1+4]
```

  - Register

```
    LDR  R0, [R1, R2]   @ mem[R1+R2]
```

  - Scaled   **@ mem[R1+4*R2]**

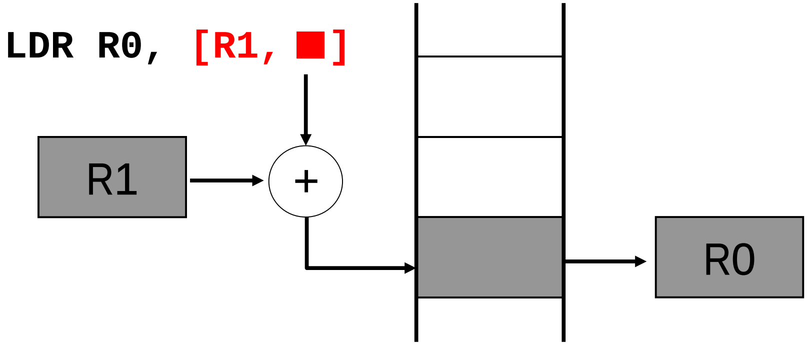
```
    LDR  R0, [R1, R2, LSL #2]
```

Addressing modes

- Pre-indexed addressing (`LDR R0, [R1, #4]`) without a writeback
- Auto-indexing addressing (`LDR R0, [R1, #4]!`) calculation before accessing with a writeback
- Post-indexed addressing (`LDR R0, [R1], #4`) calculation after accessing with a writeback

Pre-indexed addressing

```
LDR   R0, [R1, #4]      @ R0=mem[R1+4]
                        @ R1 unchanged
```

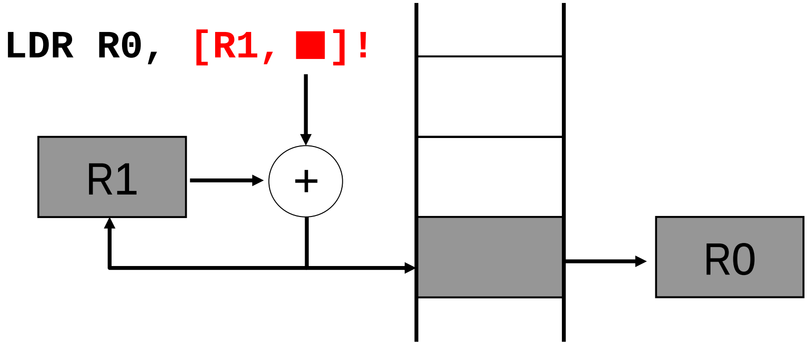


LDR R0, [R1, ■]

Auto-indexing addressing

```
LDR   R0, [R1, #4]!   @ R0=mem[R1+4]
                      @ R1=R1+4
```

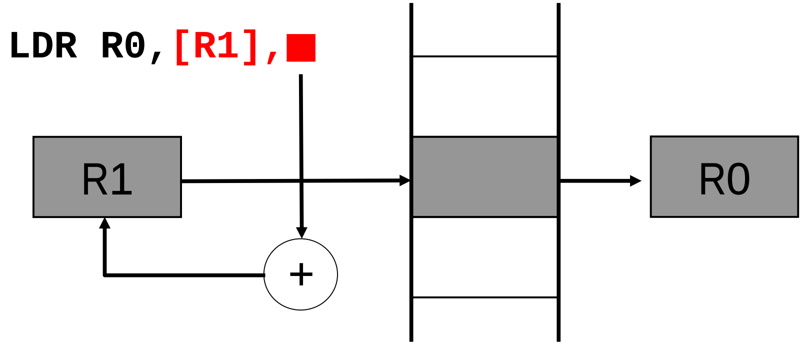
No extra time; Fast;



LDR R0, [R1, ■]!

Post-indexed addressing

```
LDR   R0, R1, #4        @ R0=mem[R1]
                        @ R1=R1+4
```



LDR R0,[R1],■

Comparisons

- Pre-indexed addressing

```
LDR  R0, [R1, R2]  @ R0=mem[R1+R2]
                   @ R1 unchanged
```

- Auto-indexing addressing

```
LDR  R0, [R1, R2]! @ R0=mem[R1+R2]
                   @ R1=R1+R2
```

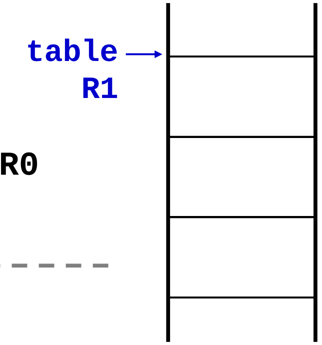
- Post-indexed addressing

```
LDR  R0, [R1], R2  @ R0=mem[R1]
                   @ R1=R1+R2
```

Application

```
         ADR R1, table
loop:    LDR R0, [R1]
         ADD R1, R1, #4
         @ operations on R0
         …
-------------------------------
         ADR R1, table
loop:    LDR R0, [R1], #4

         @ operations on R0
         …
```



Multiple register load/store

- Transfer large quantities of data more efficiently.
- Used for procedure entry and exit for saving and restoring workspace registers and the return address

registers are arranged an in increasing order; see manual

```
LDMIA  R1, {R0, R2, R5} @ R0 = mem[R1]
                        @ R2 = mem[r1+4]
                        @ R5 = mem[r1+8]
```

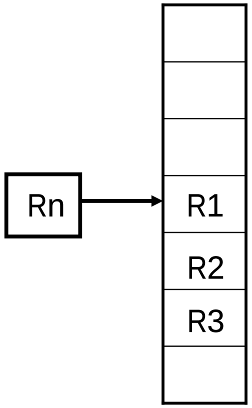
Multiple load/store register

```
LDM   load multiple registers
STM   store multiple registers
```

suffix	meaning
IA	increase after
IB	increase before
DA	decrease after
DB	decrease before

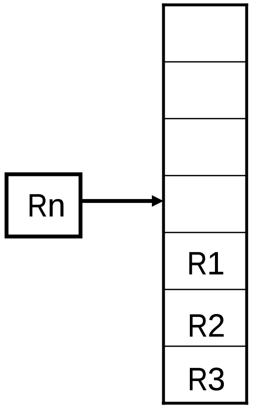
Multiple load/store register

```
LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#<registers>*4+4
DB: addr:=Rn-#<registers>*4
For each Ri in <registers>
  IB: addr:=addr+4
  DB: addr:=addr-4
  Ri:=M[addr]
  IA: addr:=addr+4
  DA: addr:=addr-4
<!>: Rn:=addr
```



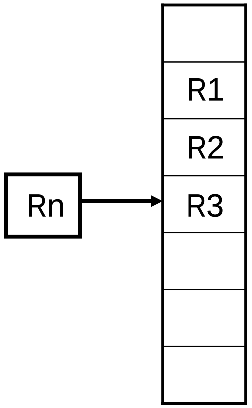
Multiple load/store register

```
LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#<registers>*4+4
DB: addr:=Rn-#<registers>*4
For each Ri in <registers>
  IB: addr:=addr+4
  DB: addr:=addr-4
  Ri:=M[addr]
  IA: addr:=addr+4
  DA: addr:=addr-4
<!>: Rn:=addr
```



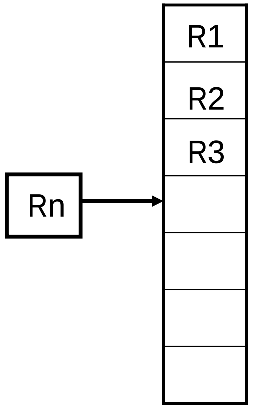
Multiple load/store register

```
LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#<registers>*4+4
DB: addr:=Rn-#<registers>*4
For each Ri in <registers>
  IB: addr:=addr+4
  DB: addr:=addr-4
  Ri:=M[addr]
  IA: addr:=addr+4
  DA: addr:=addr-4
<!>: Rn:=addr
```



Multiple load/store register

```
LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#<registers>*4+4
DB: addr:=Rn-#<registers>*4
For each Ri in <registers>
  IB: addr:=addr+4
  DB: addr:=addr-4
  Ri:=M[addr]
  IA: addr:=addr+4
  DA: addr:=addr-4
<!>: Rn:=addr
```



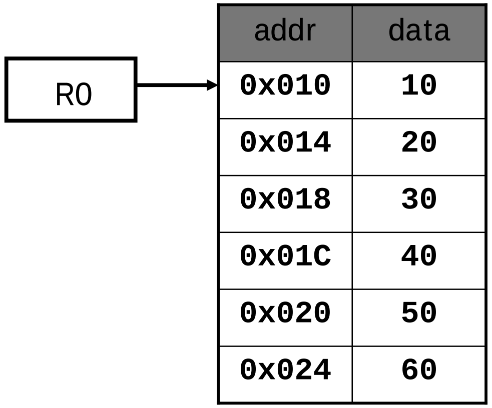
Multiple load/store register

```
LDMIA R0, {R1,R2,R3}
```

or

```
LDMIA R0, {R1-R3}
```

```
R1: 10
R2: 20
R3: 30
R0: 0x10
```

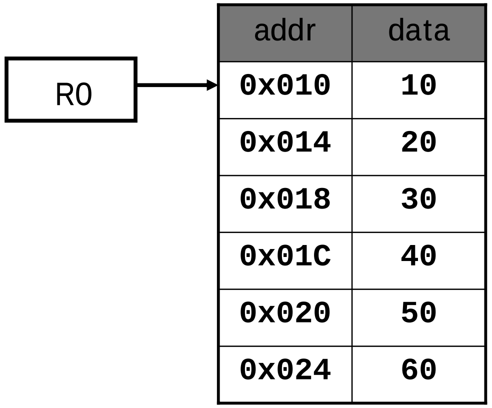


addr	data
0x010	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60

Multiple load/store register

```
LDMIA R0!, {R1,R2,R3}
```

```
R1: 10
R2: 20
R3: 30
R0: 0x01C
```

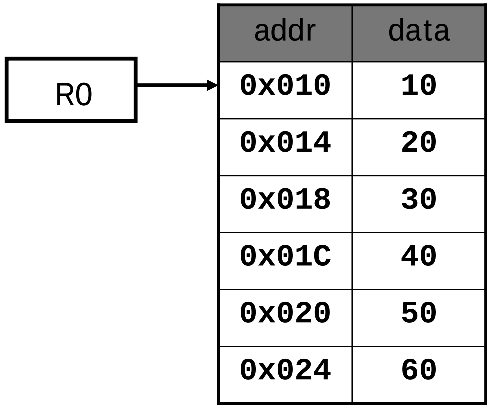


addr	data
0x010	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60

Multiple load/store register

```
LDMIB R0!, {R1,R2,R3}
```

```
R1: 20
R2: 30
R3: 40
R0: 0x01C
```

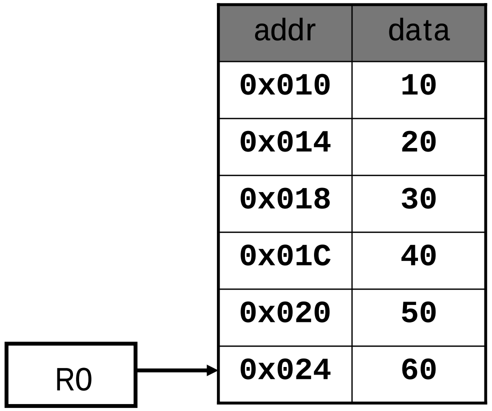


addr	data
0x010	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60

Multiple load/store register

```
LDMDA R0!, {R1,R2,R3}
```

```
R1: 40
R2: 50
R3: 60
R0: 0x018
```

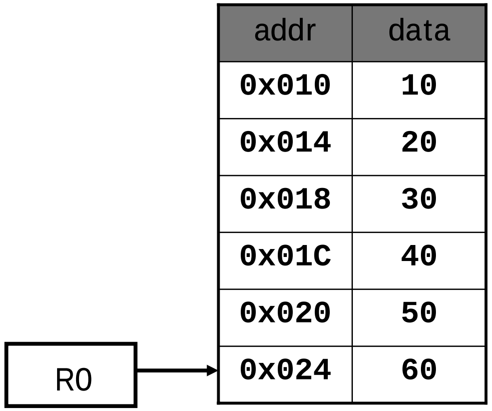


addr	data
0x010	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60

Multiple load/store register

```
LDMDB R0!, {R1,R2,R3}
```

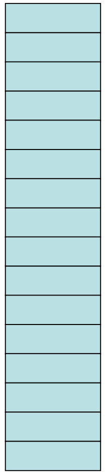
```
R1: 30
R2: 40
R3: 50
R0: 0x018
```



addr	data
0x010	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60

Application

- Copy a block of memory
  - R9: address of the source
  - R10: address of the destination
  - R11: end address of the source



```
loop:    LDMIA R9!, {R0-R7}
         STMIA R10!, {R0-R7}
         CMP   R9, R11
         BNE   loop
```

Application

- Stack (full: pointing to the last used; ascending: grow towards increasing memory addresses)

mode	POP	=LDM	PUSH	=STM
Full ascending (**FA**)	**LDMFA**	**LDMDA**	**STMFA**	**STMIB**
Full descending (**FD**)	**LDMFD**	**LDMIA**	**STMFD**	**STMDB**
Empty ascending (**EA**)	**LDMEA**	**LDMDB**	**STMEA**	**STMIA**
Empty descending (**ED**)	**LDMED**	**LDMIB**	**STMED**	**STMDA**

```
LDMFD R13!, {R2-R9}
… @ modify R2-R9
STMFD R13!, {R2-R9}
```

Control flow instructions

- Determine the instruction to be executed next
- Branch instruction

```
         B  label
         …
label:   …
```

- Conditional branches

```
         MOV  R0, #0
loop:         …
         ADD  R0, R0, #1
         CMP  R0, #10
         BNE  loop
```

Branch conditions

Branch	Interpretation	Normal uses
B BAL	Unconditional Always	Always take this branch 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 BLO	Carry clear Lower	Arithmetic operation did not give carry-out Unsigned comparison gave lower
BCS BHS	Carry set Higher or same	Arithmetic operation gave carry-out 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

Branch and link

- **BL** instruction save the return address to **R14** (lr)

```
     BL    sub    @ call sub
     CMP   R1, #5 @ return to here
     MOVEQ R1, #0
     …
sub: …            @ sub entry point
     …
     MOV   PC, LR @ return
```

Branch and link

```
        BL    sub1     @ call sub1
        …
```

use stack to save/restore the return address and registers

```
sub1:   STMFD R13!, {R0-R2,R14}
        BL    sub2
        …
        LDMFD R13!, {R0-R2,PC}

sub2:   …
        …
        MOV   PC, LR
```

Conditional execution

- Almost all ARM instructions have a condition field which allows it to be executed conditionally.

```
        movcs R0, R1
```

Mnemonic	Condition	Mnemonic	Condition
CS	*Carry Set*	CC	*Carry Clear*
EQ	*Equal* (Zero Set)	NE	*Not Equal* (Zero Clear)
VS	*Overflow Set*	VC	*Overflow Clear*
GT	*Greater Than*	LT	*Less Than*
GE	*Greater Than or Equal*	LE	*Less Than or Equal*
PL	*Plus* (Positive)	MI	*Minus* (Negative)
HI	*Higher Than*	LO	*Lower Than* (aka CC)
HS	*Higher or Same* (aka CS)	LS	*Lower or Same*

Conditional execution

```
        CMP   R0, #5
        BEQ   bypass      @ if (R0!=5) {
        ADD   R1, R1, R0 @  R1=R1+R0-R2
        SUB   R1, R1, R2 @ }
bypass: …
```

```
        CMP   R0, #5
        ADDNE R1, R1, R0
        SUBNE R1, R1, R2
```

smaller and faster

Rule of thumb: if the conditional sequence is three instructions or less, it is better to use conditional execution than a branch.

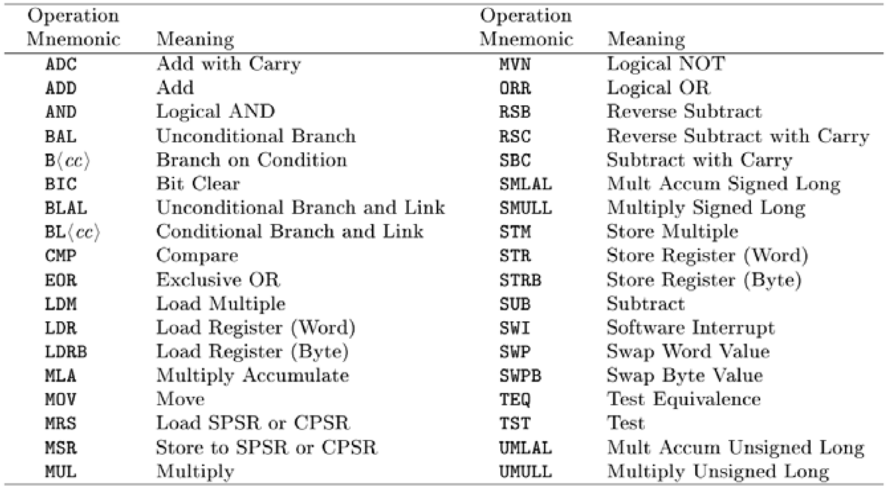
Conditional execution

```
  if ((R0==R1) && (R2==R3)) R4++
```

```
         CMP    R0, R1
         BNE    skip
         CMP    R2, R3
         BNE    skip
         ADD    R4, R4, #1
skip:    …
```

```
         CMP    R0, R1
         CMPEQ  R2, R3
         ADDEQ  R4, R4, #1
```

Instruction set



Operation Mnemonic	Meaning	Operation Mnemonic	Meaning
ADC	Add with Carry	MVN	Logical NOT
ADD	Add	ORR	Logical OR
AND	Logical AND	RSB	Reverse Subtract
BAL	Unconditional Branch	RSC	Reverse Subtract with Carry
B⟨*cc*⟩	Branch on Condition	SBC	Subtract with Carry
BIC	Bit Clear	SMLAL	Mult Accum Signed Long
BLAL	Unconditional Branch and Link	SMULL	Multiply Signed Long
BL⟨*cc*⟩	Conditional Branch and Link	STM	Store Multiple
CMP	Compare	STR	Store Register (Word)
EOR	Exclusive OR	STRB	Store Register (Byte)
LDM	Load Multiple	SUB	Subtract
LDR	Load Register (Word)	SWI	Software Interrupt
LDRB	Load Register (Byte)	SWP	Swap Word Value
MLA	Multiply Accumulate	SWPB	Swap Byte Value
MOV	Move	TEQ	Test Equivalence
MRS	Load SPSR or CPSR	TST	Test
MSR	Store to SPSR or CPSR	UMLAL	Mult Accum Unsigned Long
MUL	Multiply	UMULL	Multiply Unsigned Long

ARM assembly program

label operation operand comments

```
main:

         LDR    R1, value      @ load value
         STR    R1, result
         SWI    #11

value:   .word  0x0000C123
result:  .word  0
```

Shift left one bit

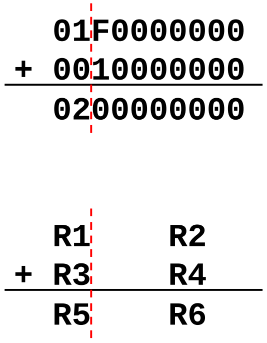
```
         ADR  R1, value
         MOV  R1, R1, LSL #0x1
         STR  R1, result
         SWI  #11
value:   .word  4242
result:  .word  0
```

Add two numbers

```
main:
         ADR  R1, value1
         ADR  R2, value2
         ADD  R1, R1, R2
         STR  R1, result
         SWI  #11
value1:  .word  0x00000001
value2:  .word  0x00000002
result:  .word  0
```

64-bit addition

```
       ADR  R0, value1
       LDR  R1, [R0]
       LDR  R2, [R0, #4]
       ADR  R0, value2
       LDR  R3, [R0]
       LDR  R4, [R0, #4]
       ADDS R6, R2, R4
       ADC  R5, R1, R3
       STR  R5, [R0]
       STR  R6, [R0, #4]
       SWI  #11
value1: .word  0x00000001, 0xF0000000
value2: .word  0x00000000, 0x10000000
result: .word  0
```



```
   01F0000000
 + 0010000000
 ------------
   0200000000

   R1     R2
 + R3     R4
 ------------
   R5     R6
```

Loops

- For loops

```
for (i-0; i<10; i++) {a[i]=0;}

      MOV  R1, #0
      ADR  R2, A
      MOV  R0, #0
LOOP: CMP  R0, #10
      BGE  EXIT
      STR  R1, [R2, R0, LSL #2]
      ADD  R0, R0, #1
      B    LOOP
EXIT:  ..
```

Loops

- While loops

```
LOOP: …    ; evaluate expression
      BEQ  EXIT
      …   ; loop body
      B    LOOP
EXIT: …
```

Find larger of two numbers

```
        ADR  R1, value1
        ADR  R2, value2
        CMP  R1, R2
        BHI  Done
        MOV  R1, R2
Done:
        STR  R1, result
        SWI  #11
value1: .word  4
value2: .word  9
result: .word  0
```

GCD

```
int gcd (int I, int j)
{
    while (i!=j)
    {
      if (i>j)
        i -= j;
      else
        j -= i;
    }
}
```

GCD

```
Loop:  CMP   R1, R2
       SUBGT R1, R1, R2
       SUBLT R2, R2, R1
       BNE   loop
```

Count negatives

```
; count the number of negatives in
; an array DATA of length LENGTH

     ADR   R0, DATA     @ R0 addr
     EOR   R1, R1, R1   @ R1 count
     LDR   R2, Length   @ R2 index
     CMP   R2, #0
     BEQ   Done
```

Count negatives

```
loop:
     LDR   R3, [R0]
     CMP   R3, #0
     BPL   looptest
     ADD   R1, R1, #1 @ it’s neg.
looptest:
     ADD   R0, R0, #4
     SUBS  R2, R2, #1
     BNE   loop
```

Subroutines

- Passing parameters in registers

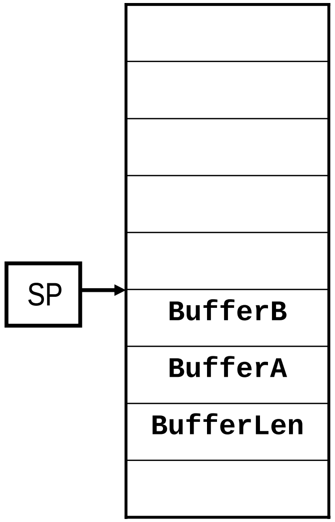
  Assume that we have three parameters **BufferLen, BufferA, BufferB** to pass into a subroutine

```
ADR  R0, BufferLen
ADR  R1, BufferA
ADR  R2, BufferB
BL   Subr
```

Passing parameters using stacks

- Caller

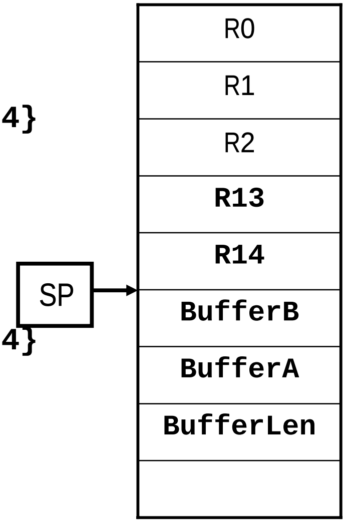
```
MOV  R0, #BufferLen
STR  R0, [SP, #-4]!
MOV  R0, =BufferA
STR  R0, [SP, #-4]!
MOV  R0, =BufferB
STR  R0, [SP, #-4]!
BL   Subr
```



Passing parameters using stacks

- Callee

```
Subr
 STMDB SP, {R0,R1,R2,R13,R14}
 LDR   R2, [SP, #0]
 LDR   R1, [SP, #4]
 LDR   R0, [SP, #8]
  …
 LDMDB SP, {R0,R1,R2,R13,R14}
 MOV   PC, LR
```



SP

R0

R1

R2

R13

R14

BufferB

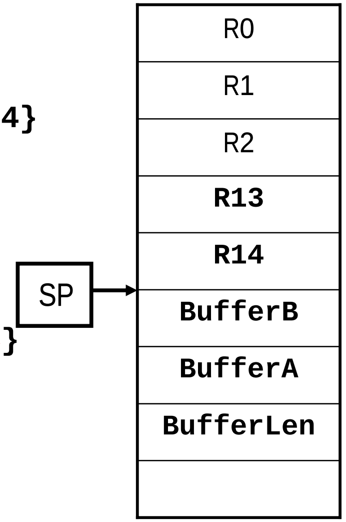
BufferA

BufferLen

Passing parameters using stacks

- Callee

```
Subr
 STMDB SP, {R0,R1,R2,R13,R14}
 LDR   R2, [SP, #0]
 LDR   R1, [SP, #4]
 LDR   R0, [SP, #8]
  …
 LDMDB SP, {R0,R1,R2,R13,PC}
```



SP

R0

R1

R2

R13

R14

BufferB

BufferA

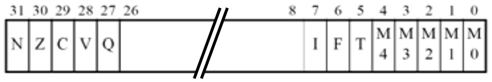
BufferLen

Review

- ARM architecture
- ARM programmer model
- ARM instruction set
- ARM assembly programming

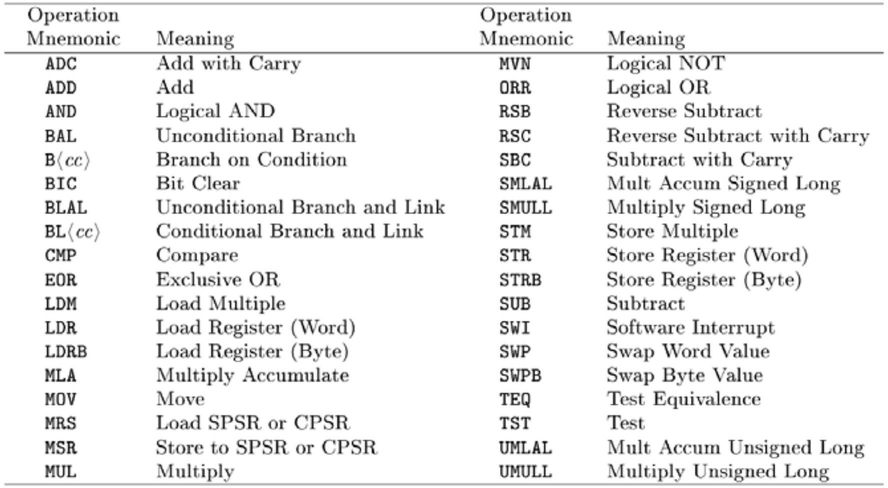
ARM programmer model

R0	R1	R2	R3
R4	R5	R6	R7
R8	R9	R10	R11
R12	R13	R14	PC



0x00000000	00
0x00000001	10
0x00000002	20
0x00000003	30
0x00000004	FF
0x00000005	FF
0x00000006	FF
0xFFFFFFFD	00
0xFFFFFFFE	00
0xFFFFFFFF	00

Instruction set



Operation Mnemonic	Meaning	Operation Mnemonic	Meaning
ADC	Add with Carry	MVN	Logical NOT
ADD	Add	ORR	Logical OR
AND	Logical AND	RSB	Reverse Subtract
BAL	Unconditional Branch	RSC	Reverse Subtract with Carry
B⟨cc⟩	Branch on Condition	SBC	Subtract with Carry
BIC	Bit Clear	SMLAL	Mult Accum Signed Long
BLAL	Unconditional Branch and Link	SMULL	Multiply Signed Long
BL⟨cc⟩	Conditional Branch and Link	STM	Store Multiple
CMP	Compare	STR	Store Register (Word)
EOR	Exclusive OR	STRB	Store Register (Byte)
LDM	Load Multiple	SUB	Subtract
LDR	Load Register (Word)	SWI	Software Interrupt
LDRB	Load Register (Byte)	SWP	Swap Word Value
MLA	Multiply Accumulate	SWPB	Swap Byte Value
MOV	Move	TEQ	Test Equivalence
MRS	Load SPSR or CPSR	TST	Test
MSR	Store to SPSR or CPSR	UMLAL	Mult Accum Unsigned Long
MUL	Multiply	UMULL	Multiply Unsigned Long

References

- ARM Limited. ARM Architecture Reference Manual.
- Peter Knaggs and Stephen Welsh, *ARM: Assembly Language Programming*.
- Peter Cockerell, ARM Assembly Language Programming.
- Peng-Sheng Chen, *Embedded System Software Design and Implementation.*