ARM Instruction Set

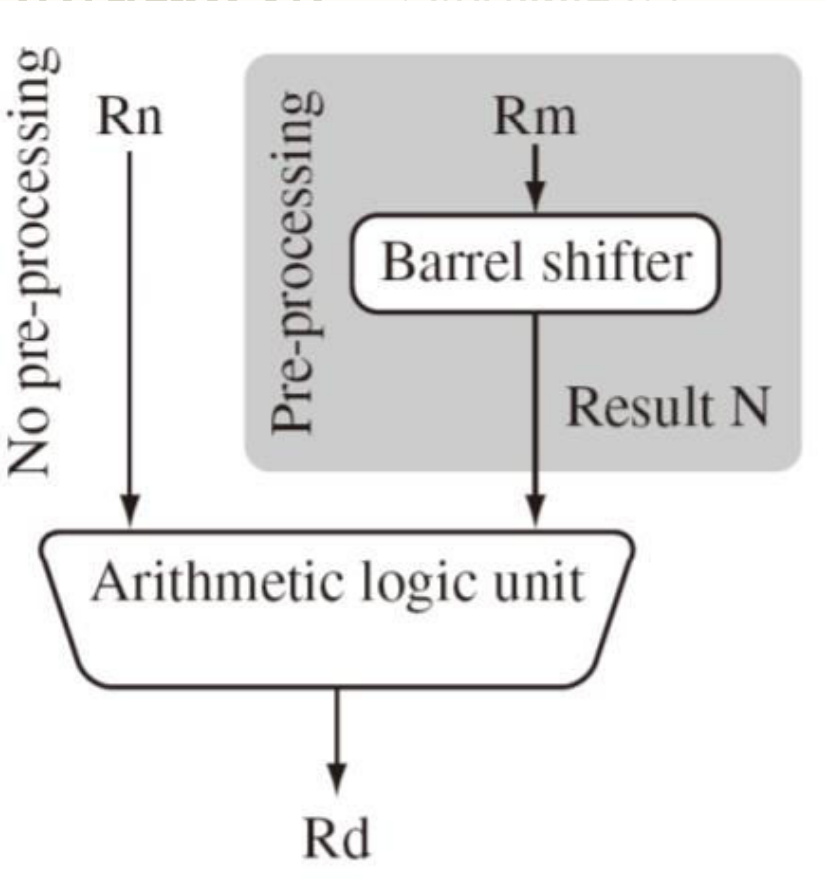
ARM Instruction Set

- Instructions process data held in registers and access memory with load and store instructions.

- Classification of instructions

  - Data processing instructions
  - Data Transfer Instructions - Load-Store instructions
  - Control flow instructions – Conditional & Branching instructions
  - S/W interrupt instructions
  - Program status register instructions

1. Data Processing Instructions



- Perform arithmetic, logical, move, compare and multiply operations
- All operations except multiply instructions are carried out in ALU
- Uses 3- address instruction format:
  - Eg: ADD R0,R1,R2 ; R0 = R1+R2
- All operands are 32 bits wide.
  - From register.
  - Specified as literals in instruction itself.
- Result is 32 bits wide
  - Stored in register file.
- Most Data Processing instructions can Pre-process one operand using barrel shifter.

  Eg: ADD R0,R1,R2, LSL #2 ; R2 = LSL by 2 position
  ; R0 = R1+R2

1. Data processing

- Manipulate data within register

- 1.1 Arithmetic instructions
- 1.2 Logical instructions
- 1.3 Move instructions
- 1.4 Comparison instruction

1.1 Arithmetic Instructions

- ADD r0,r1,r2 – ADD *add*
- ADC r0,r1,r2 – ADC *add with carry*
- SUB r0,r1,r2 – SUB *subtract*
- SBC r0,r1,r2 – SBC *subtract with carry*
- RSB r0,r1,r2 – RSB *reverse subtract*
- RSC r0,r1,r2 – RSC *reverse subtract with carry*

- *Carry-in is the current value of C bit in CPSR*

Data processing instructions

Arithmetic Operations

```
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
```

' ADD ' is simple addition, ' ADC ' is add with carry,
' SUB ' is subtract, ' SBC ' is subtract with carry,
' RSB ' is reverse subtraction and ' RSC ' reverse subtract with carry.

1.2 Bit-wise logical operations

- AND r0,r1,r2
- ORR r0,r1,r2
- EOR r0,r1,r2
- BIC r0,r1,r2

- BIC
  - *Bit Clear*
  - Every ‘1’ in the second operand clears the corresponding bit in the first.

Data processing instructions

Bit-wise 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
BIC     r0, r1, r2          ; r0 := r1 and not r2
```

perform the specified Boolean logic operation on each bit pair
of the input operands, so in the first case r0[i]:= r1[i] AND r2[i] for each value
of i from 0 to 31 inclusive, where r0[i] is the ith bit of r0.

BIC, stands for'bit clear' where every ' 1' in the second operand clears the corresponding bit in the first.

1.3 Register- Register movement operations

MOV Rd, N

Rd :destination register

N: can be an immediate value or source register

Example: MOV r7,r5

MVN Rd, N

Move into Rd negated value of the 32 bit value from source (N').

Example: MVN r7,r5

Data processing instructions

Register movement operations

```
MOV     r0, r2                  ; r0 := r2
MVN     r0, r2                  ; r0 := not r2
```

- The ' MVN ' mnemonic stands for 'move negated'
  - **inverting** every bit in the source operand.

1.4 Comparison

- Enables comparison of 32 bit values
  - Updates CPSR flags but do not affect other registers

CMP r1,r2 *compare*

CMN r1,r2 *compare negated*

Data processing instructions

Comparison operations

- set the condition code bits (N, Z, C and V) in the CPSR according to the selected operation.

```
CMP  r1, r2          ; set cc on r1 - r2
CMN  r1, r2          ; set cc on r1 + r2
TST  r1, r2          ; set cc on r1 and r2
TEQ  r1, r2          ; set cc on r1 xor r2
```

- The mnemonics stand for 'compare' (CMP), 'compare negated' (CMN), '(bit) test' (TST) and 'test equal' (TEQ).

Data processing instructions

Immediate operands

ADD r3, r3, #1 ----------→ r3: =r3 + 1

- **Specifying immediate operands:**

```
ADD    r1,r2,#2          ; r1 = r2 + 2
SUB    r3,r3,#1          ; r3 = r3 - 1
AND    r6,r4,#&0f        ; r6 = r4[3:0]
```

- Notations:
  - # indicates immediate value
  - & indicates hexadecimal notation
- Allowed immediate values:
  - 0 to 255 (8 bits), rotated by any number of bit positions that is multiple of 2.

Video player

Shifted Register Operands

ADD r3,r2,r1, LSL #3

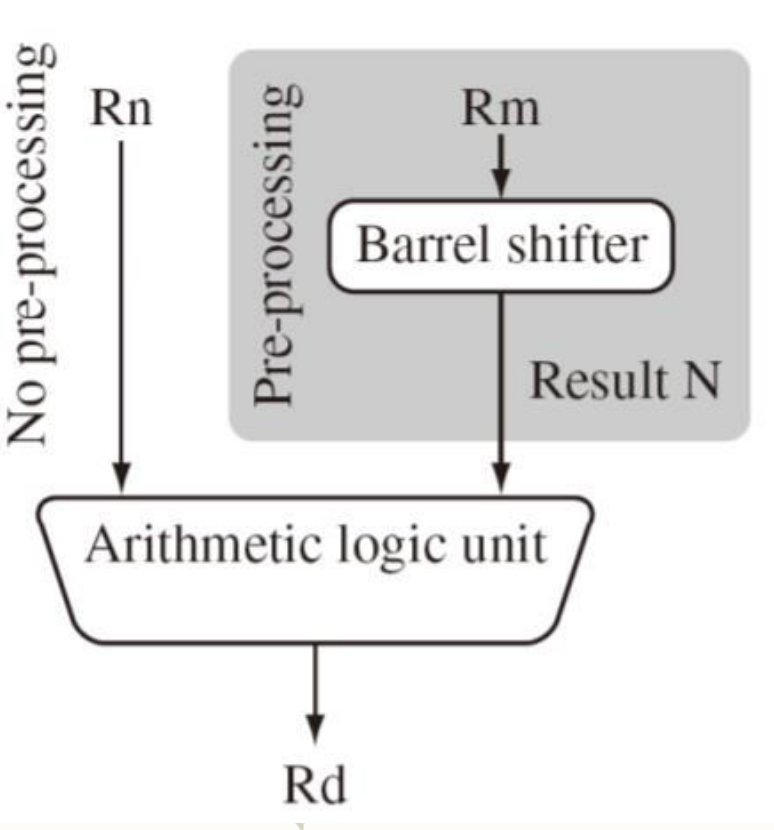
- Single instruction.
- Second reg operand subjected to shift.
- LSL : Logical Shift Left.
- Various shift & rotate operations:
  - Logical Shift Left (LSL),
  - Logical Shift Right (LSR),
  - Arithmetic Shift Left (ASL),
  - Arithmetic Shift Right (ASR),
  - Rotate Right (ROR),
  - Rotate Right Extended (RRX)

- **Shifted register operands:**
  - The second source operand may be shifted either by a constant number of bit positions, or by a register-specified number of bit positions.

```
ADD    r1,r2,r3,LSL #3    ; r1 = r2 + (r3 << 3)
ADD    r1,r2,r3,LSL r5    ; r1 = r2 + (r3 << r5)
```

- Various shift and rotate options:
  - **LSL**: logical shift left
  - **LSR**: logical shift right
  - **ROR**: rotate right
  - **RRX**: rotate right extended by 1 bit
  - **ASL**: arithmetic shift left
  - **ASR**: arithmetic shift right

With Barrel shifter



- Use of Barrel shifter with arithmetic and logical instructions increases the set of possible available operations.
  - Useful for fast multiplication and dealing with lists, table and other complex data structure.

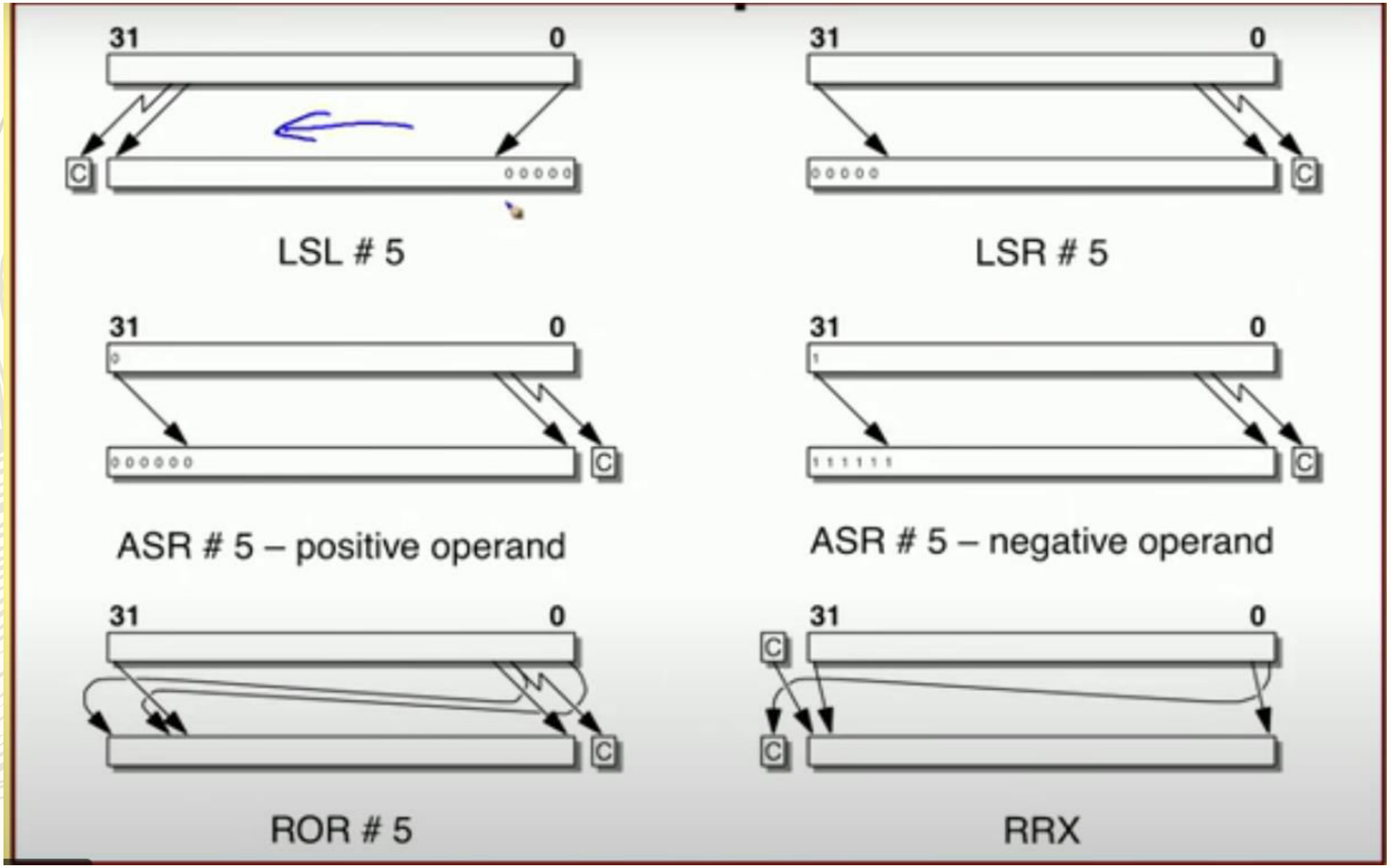
- the second register operand to be subject to a shift operation before it is combined with the first operand.

```
ADD r3,  r2,  r1,  LSL #3 ;
```

r3 := r2 + 8 x r1

It is still a single ARM instruction, executed in a single clock cycle.

ARM combines shift operations with a general ALU operation in a single instruction


31
0
C
00000
LSL # 5
31
0
00000
C
LSR # 5
31
0
0
000000
C
ASR # 5 – positive operand
31
0
1
111111
C
ASR # 5 – negative operand
31
0
C
ROR # 5
31
0
C
C
RRX

Assembler Mnemonic	OpCode	Action
AND	0000	operand1 AND operand2
EOR	0001	operand1 EOR operand2
SUB	0010	operand1 - operand2
RSB	0011	operand2 - operand1
ADD	0100	operand1 + operand2
ADC	0101	operand1 + operand2 + carry
SBC	0110	operand1 - operand2 + carry - 1
RSC	0111	operand2 - operand1 + carry - 1
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	operand1 OR operand2
MOV	1101	operand2 (operand1 is ignored)
BIC	1110	operand1 AND NOT operand2 (Bit clear)
MVN	1111	NOT operand2 (operand1 is ignored)

Multiply instructions

- Multiply contents of a pair of registers
  - Multiplication by immediate constants not supported.
    - long multiply generates 64 bit result

– Examples:

– → mul r0,r1,r2

– Contents of r1 and r2 multiplied and the result is put in r0.

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

        MUL   r0, r1, r2    ; r0 = r1*r2

POST    r0 = 0x00000004
        r1 = 0x00000002
        r2 = 0x00000002
```

```
PRE     r0 = 0x00000000
        r1 = 0x00000000
        r2 = 0xf0000002
        r3 = 0x00000002

        UMULL   r0, r1, r2, r3   ; [r1,r0] = r2*r3

POST    r0 = 0xe0000004 ; = RdLo
        r1 = 0x00000001 ; = RdHi
```

– →umull r0,r1,r2,r3 ; *unsigned multiply long*

– Unsigned multiplication with result stored in r0 and r1.

– Number of cycles taken for execution of multiplication instruction depends upon processor implementation.

- **Multiplication instruction**

```
MUL    r1,r2,r3              ; r1 = (r2 x r3)[31:0]
```

  - Only the least significant 32-bits are returned.
  - Immediate operands are not supported.

- **Multiply-accumulate instruction:**

```
MLA    r1,r2,r3,r4           ; r1 = (r2 x r3 + r4)[31:0]
```

  - Required in digital signal processing (DSP) applications.

- Multiplication with 64-bit results is also supported.

Multiply and Accumulate

- Result of multiplication can be accumulated with content of another register

  - MLA Rd, Rn, Rm, Racc
  - Rd = (Rn*Rm)+Racc

  - UMLAL RdLo, RdHi, Rn, Rm
  - [RdHi,RdLo]=[RdHi,RdLo]+(Rn*Rm)

(b) Data Transfer Instructions

- ARM instruction set supports three types of data transfers:
  a) Single register loads and stores
     - Flexible, supports byte, half-word and word transfers
  b) Multiple register loads and stores
     - Less flexible, multiple words, higher transfer rate
  c) Single register-memory swap
     - Mainly for system use (for implementing locks)

Data Transfer Instructions

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*
LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*

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

- Transfer data between memory and processor registers

➢ Single register load/store

– Data types supported are signed and unsigned words(32 bits), half word, bytes. Eg.

```
LDR  R0, [R1]    @ R0 := mem32[R1]
STR  R0, [R1]    @ mem32[R1] := R0
```

➢ Multiple register load/store

– Transfer multiple registers between memory and the processor in a single instruction

➢ Swap

– swaps content of a memory location with the contents of a register. For semaphore implementation. Eg. SWP (B)

- All ARM data transfer instructions use *register indirect addressing*.
  - Before any data transfer, some register must be initialized with a memory address.

```
ADRL   r1,Table      ; r1 = memory address of Table
```

  - Example:

```
LDR    r0,[r1]       ; r0 = mem[r1]
STR    r0,[r1]       ; mem[r1] = r0
```

- **Single register loads and stores**
  - The simplest form uses register indirect without any offset:

```
LDR   r0,[r1]      ; r0 = mem[r1]
STR   r0,[r1]      ; mem[r1] = r0
```

  - An alternate form uses register indirect with offset (limited to 4 Kbytes):

```
LDR   r0,[r1,#4]   ; r0 = mem[r1+4]
STR   r0,[r1,#12]  ; mem[r1+12] = r0
```

  - We can also use auto-indexing in addition:

```
LDR   r0,[r1,#4]!    ; r0 = mem[r1+4],  r1 = r1 + 4
STR   r0,[r1,#12]!   ; mem[r1+12] = r0, r1 = r1 + 4
```

- We can use post indexing:

```
LDR   r0,[r1],#4   ; r0 = mem[r1],   r1 = r1 + 4
STR   r0,[r1],#12  ; mem[r1] = r0,   r1 = r1 + 12
```

- We can specify a byte or half-word to be transferred:

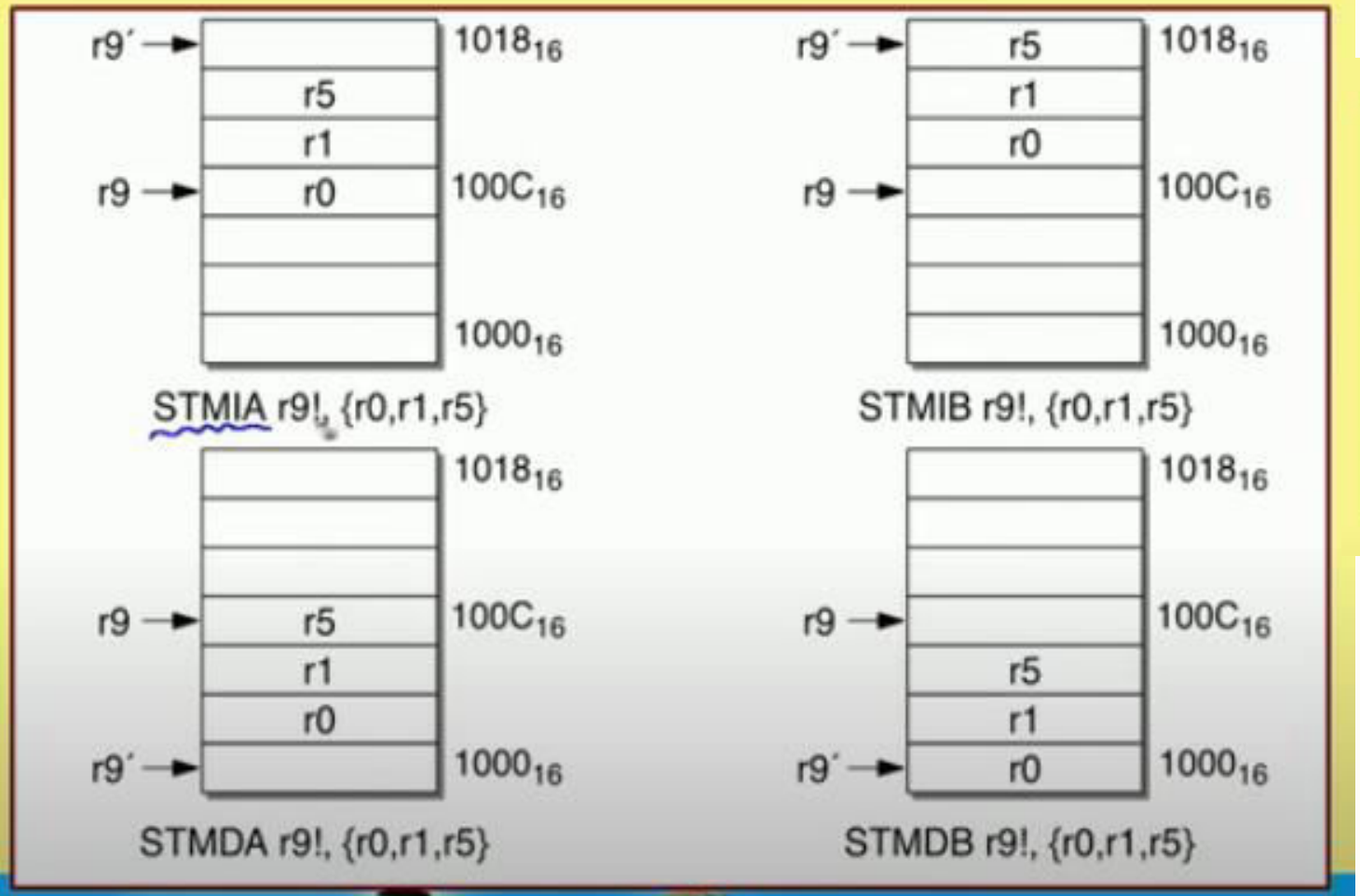
```
LDRB  r0,[r1]      ; r0 = mem8[r1]
STRB  r0,[r1]      ; mem8[r1] = r0
LDRSH r0,[r1]      ; r0 = mem16[r1]
STRSH r0,[r1]      ; mem16[r1] = r0
```

- **Multiple register loads and stores**
  - ARM supports instructions that transfer between several registers and memor
  - Example:

```
LDMIA r1,{r3,r5,r6}      ; r3 = mem[r1]
                         ; r5 = mem[r1+4]
                         ; r6 = mem[r1+8]
```

  - For **LDMIB**, the addresses will be **r1+4**, **r1+8**, **and r1+12**.
  - The list of destination registers may contain any or all of r0 to r15.
- Block copy addressing
  - Supported with addresses that can *increment (I)* or *decrement (D)*, before (B) or *after (A)* each transfer.

- Examples of addressing modes in multiple-register transfer



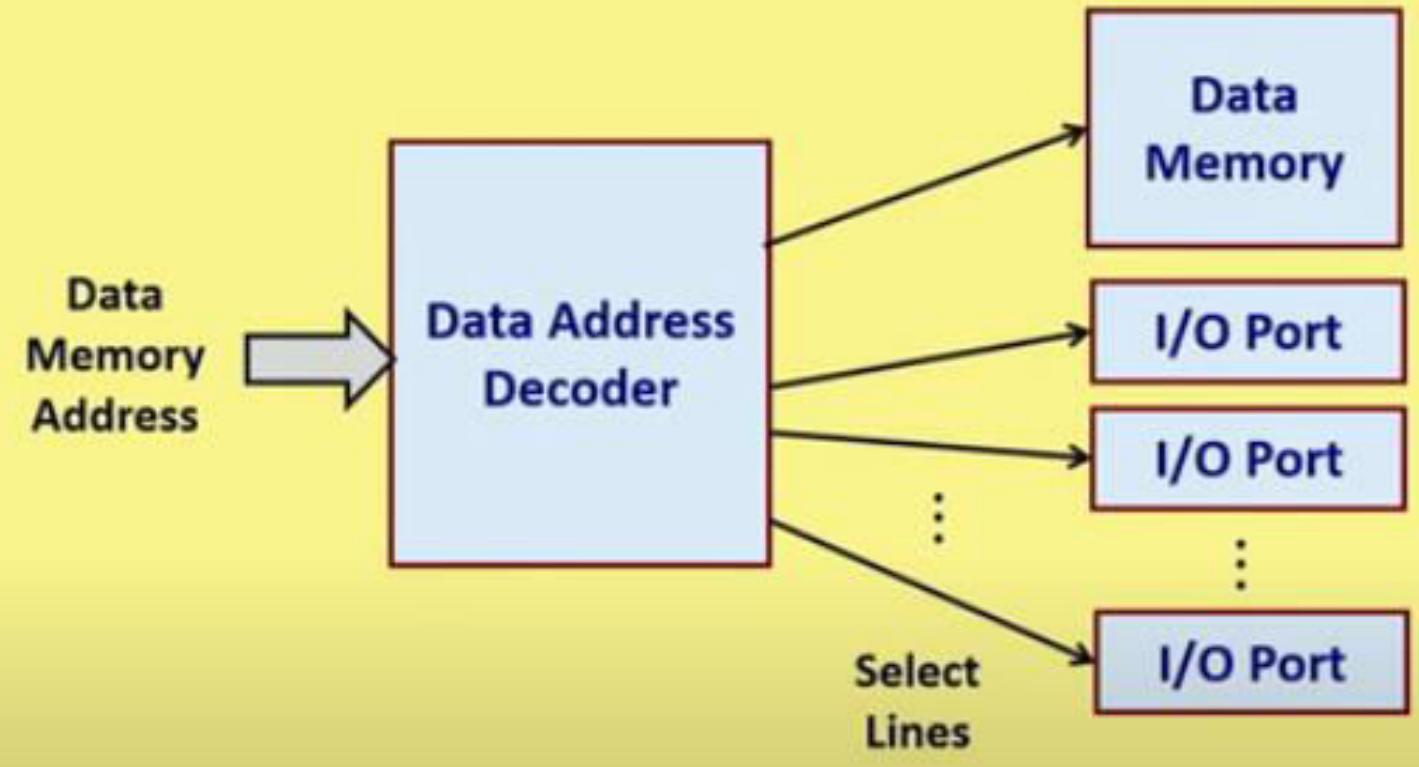
LDMIA, STMIA
- Increment after

LDMIB and STMIB
- Increment before

LDMDA, STMDA
- Decrement after

LDMDB, STMDB
- Decrement before

Memory Mapped I/O in ARM



- No separate instructions for input/output.
- The I/O ports are treated as data memory locations.
  - Each with a unique (memory) address.
- Data input is done using the LDR instruction.
- Data output is done with the STR instruction.

Control flow instructions

- Determines which instructions next.
  - Branch
  - 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

Mnemonic	Name	Condition flags
EQ	equal	*Z*
NE	not equal	*z*
CS HS	carry set/unsigned higher or same	*C*
CC LO	carry clear/unsigned lower	*c*
MI	minus/negative	*N*
PL	plus/positive or zero	*n*
VS	overflow	*V*
VC	no overflow	*v*
HI	unsigned higher	*zC*
LS	unsigned lower or same	*Z* or *c*
GE	signed greater than or equal	*NV* or *nv*
LT	signed less than	*Nv* or *nV*
GT	signed greater than	*NzV* or *nzv*
LE	signed less than or equal	*Z* or *Nv* or *nV*
AL	always (unconditional)	ignored

(c) Control Flow Instructions

- These instructions change the order of instruction execution.
  - Normal flow is sequential execution, where PC is incremented by 4 after executing every instruction.
- Types of conditional flow instructions:
  - Unconditional branch
  - Conditional branch
  - Branch and Link
  - Conditional execution

- Unconditional branch instruction:

```
                B       Target
                ...
                ...
Target          ...
```

- Conditional branch instruction:

```
                MOV     r2,#0
LOOP            ...
                ...
                ADD     r2,r2,#1
                CMP     r0,#20
                BNE     LOOP
                ...
```

- Branch conditions that are supported:
  - B, BAL — Unconditional branch
  - BEQ, BNE — Equal or not equal to zero
  - BPL, PMI — Result positive or negative
  - BCC, BCS — Carry set or clear
  - BVC, BVS — Overflow set or clear
  - BGT, BGE — Greater than, greater or equal
  - BLT, BLE — Less than, less or equal

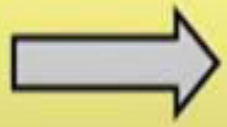
- **Branch and link instruction**
  - Used for calling subroutines in ARM.
  - The return address is saved in register r14 (called *link register*).
  - To return from the subroutine, we have to jump back to the address stored in r14.

```
        BL    MYSUB          ; Branch to subroutine
        ...                  ; Return here
        ...
MYSUB   ...                  ; Subroutine starts here
        ...
        MOV   pc,r14         ; Return
```

Nested subroutine calls cannot be used in this way.

- **Conditional execution**
  - A unique feature of the ARM instruction set.
  - All instructions can be made conditional, i.e. will get executed only when a specified condition is true.
  - Helps in removing many short branch instructions (improves performance and code density)
  - An example: `if (r2 != 10) r5 = r5 +10 - r3`

```
    CMP     r2,#10
    BEQ     SKIP
    ADD     r5,r5,r2
    SUB     r5,r5,r3
SKIP ...
```



```
CMP     r2,#10
ADDNE   r5,r5,r2
SUBNE   r5,r5,r3
```

- Various instruction postfix supported for conditional execution:

Postfix	Condition
CS	Carry set
EQ	Equal (zero set)
VS	Overflow set
GT	Greater than
GE	Greater than or equal
PL	Plus (positive)
HI	Higher than
HS	Higher or same (i.e. CS)

Postfix	Condition
CC	Carry clear
NE	Not equal (zero clear)
VC	Overflow clear
LT	Less than
LE	Less than or equal
MI	Minus (negative)
LO	Lower than (i.e. CC)
LS	Lower or same

- **Supervisor calls**
  - The *Software Interrupt Instruction (SWI)* is used to enter Supervisor Mode, usually to request a particular supervisor function (e.g. input or output).
  - The CPSR is saved into the Supervisory Mode SPSR, and execution branches to the SWI vector.
  - The SWI handler reads the opcode to extract the SWI function number.

ARM Addressing:

- Register Indirect addressing.
- Value in one register taken as memory address.
  - *LDR* $\mathbf{r_l}$*,* $\mathbf{[r_m]}$; $r_l = mem[r_m]$
  - *STR* $\mathbf{r_l}$*,* $\mathbf{[r_m]}$; $mem[r_m] = r_l$
- *Offset addressing:*
  - *Pre-indexed: LDR r0,[r1,#4]*
  - *Pre-indexed with auto-indexing: LDR r0,[r1,#4]!*
  - *Post-indexed: LDR r0,[r1],#4*

Stack Addressing:

- Full ascending
  - *Stack grows up*
  - *Base register → highest addr containing valid item.*
- Empty ascending
  - *Stack grows up*
  - *Base register → first empty location above.*
- Full descending
  - *Stack grows down*
  - *Base register → lowest addr containing valid item.*
- Empty descending
  - *Stack grows down*
  - *Base register → first empty location below.*

Thank You

References:

Text books: ARM System on Chip Architecture, 2nd edition,by Steve Furber (Chapter 3 & 5)

https://youtu.be/UdY5RkkT7bg?list=PLbRMhDVUMngcJu5oUhgpgYqtOn7DmSfuU

https://youtu.be/yXSTWW0O1LE?list=PLbRMhDVUMngcJu5oUhgpgYqtOn7DmSfuU

https://youtu.be/xYQ60EqTNuo?list=PLbRMhDVUMngcJu5oUhgpgYqtOn7DmSfuU