An Introduction to ARM Processor

Dinesh Sharma

EE Department
IIT Bombay, Mumbai

January 15, 2024

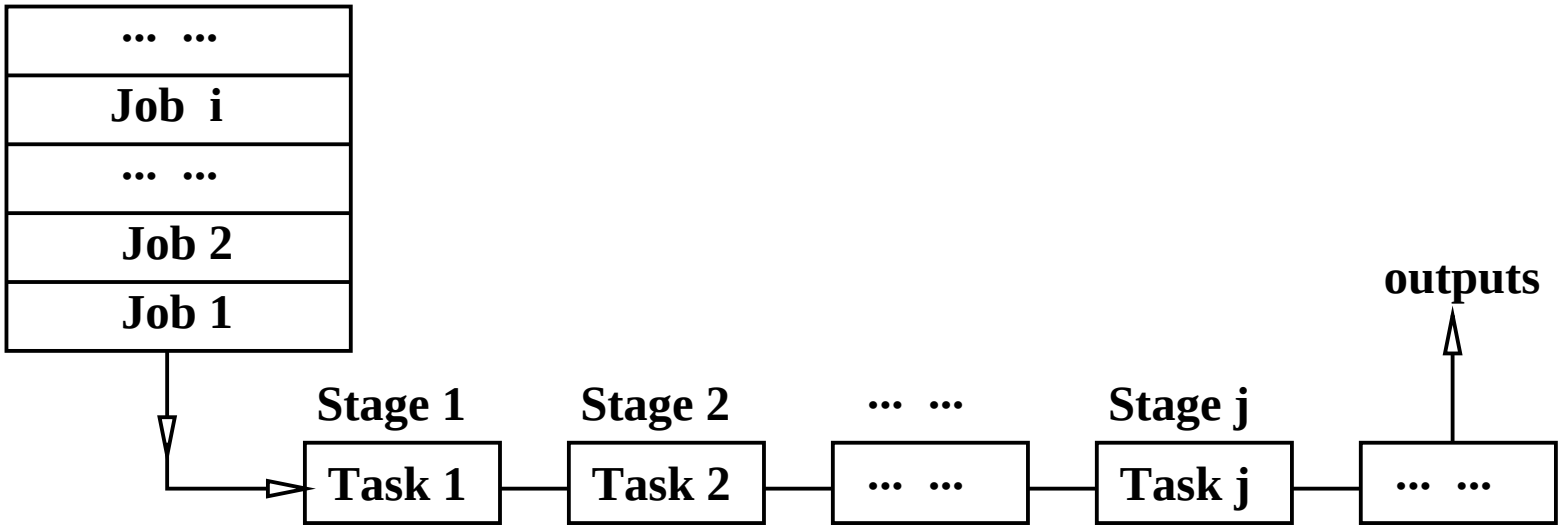
RISC architecture

- While designing the architecture of a processor, one should prefer to spend resources (silicon real estate, power . . .) on those instructions which are most frequently executed.
- By avoiding implementation of complex but infrequently used instructions, we can use the saved silicon real estate for higher number of on-chip registers and other useful design elements.
- This suggests the use of a processor with a reduced instruction set, abbreviated to RISC.
- The ARM processor is based on a RISC architecture.
- However, it deviates from a pure RISC philosophy for some instructions.

RISC architecture

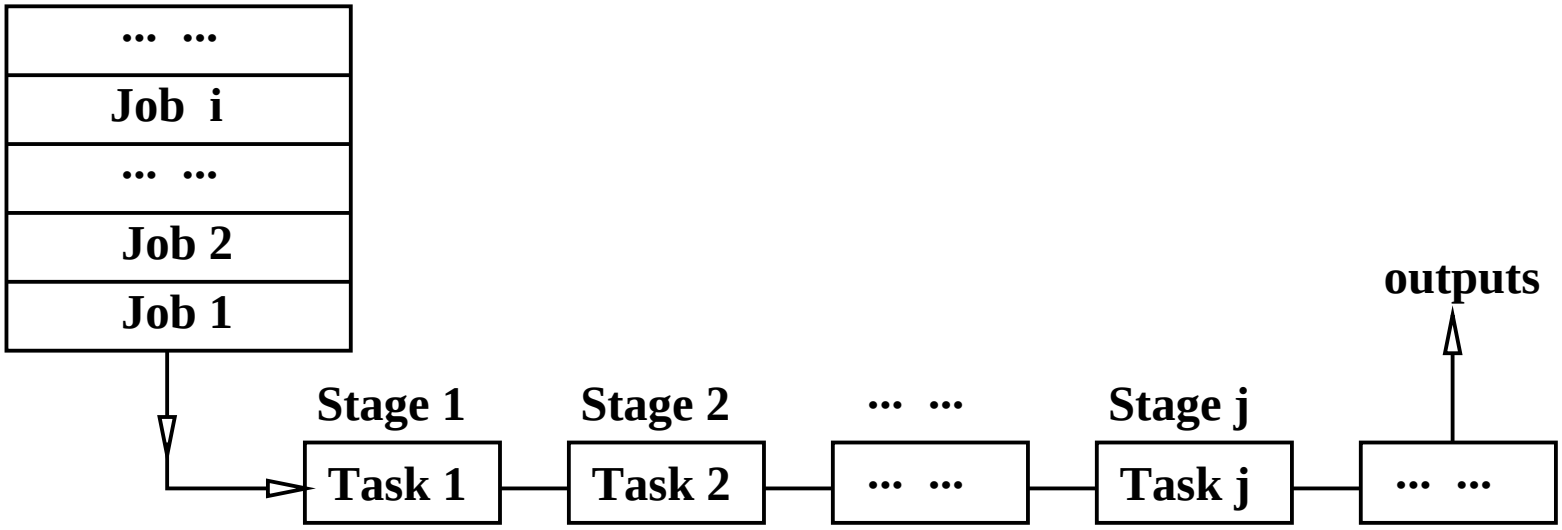
- Although by itself, the RISC architecture implies just a reduced and regular instruction set, which frees up silicon real estate for a larger number of registers and other useful hardware units like barrel shifters, most RISC processors have adopted other architectural features as well to improve performance.
- Most RISC processors are pipelined. In a pipelined architecture, the hardware works on several instructions at the same time. While one instruction is being fetched, the previous one is being decoded and the one before that is begin executed. This improves throughput.
- Most RISC processors also use a load-store architecture. This means that only the load and store instructions access the memory. All other instructions operate only on registers. If you need to operate on a memory operand, it should first be loaded into a register and then the operation is carried out.

Pipelining



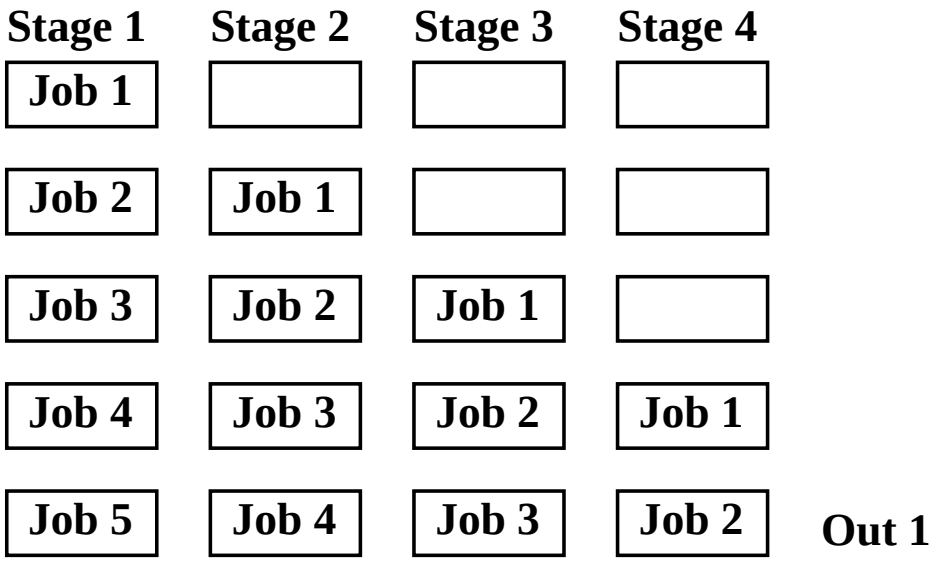
- We often have to perform several jobs repetitively, where each job requires many tasks to be performed in sequence.
- Each task may require specialised hardware which is specific to this task.
- A simple minded way to do this would be to take each job from the queue, perform all its tasks, then take up the next job and so on . . . .

Pipelining



- With the simple minded approach, only the hardware specific to the task being performed at a given time will be active, while all other stages will be idle.
- Can't we begin doing the next job using the idle hardware even before all the tasks for the first job are over?
- Indeed we can! This is called pipelining.

Pipelining



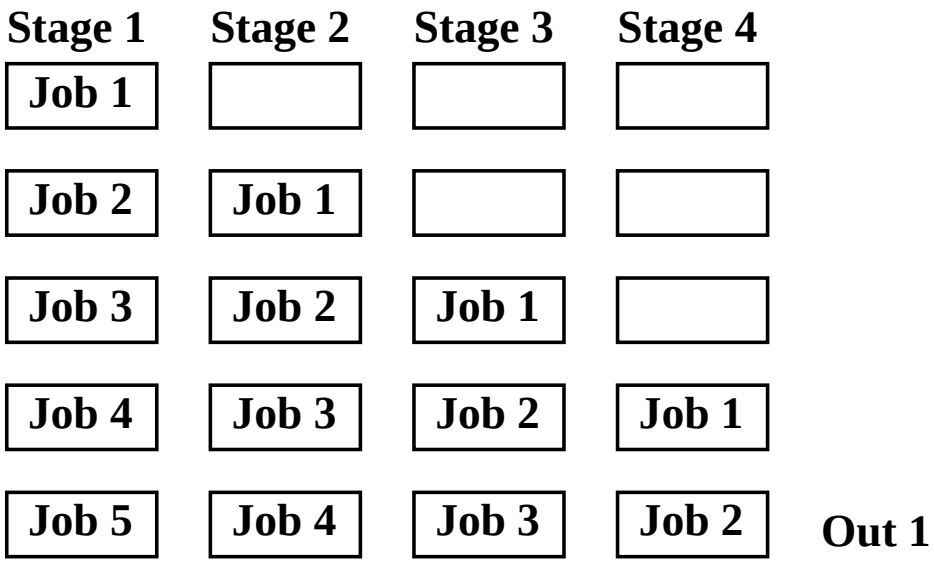
- Notice that the time between the arrival of a job and the availability of the *corresponding* output has not been reduced.
- However, new jobs can be taken up much more quickly and the outputs are generated much more frequently.

The time elapsed between the arrival of a job and the production of the corresponding output is called its latency.

The number of jobs handled per unit time is called throughput.

Pipelining improves the throughput of a repetitive process.

Pipelining



- Flow through the pipeline will be dominated by the slowest operation.
- For an optimal implementation, tasks should be so chosen so that all of them take the same amount of time.

In case of a processor, this suggests a constant sized instruction word and balanced operation time for the decode, execute and write-back stages.

Load/store Architecture

- A read or write operation on a random external memory location is extremely slow compared to a read or write operation on an internal register.
  (It may be slower by a factor of 10 or even more!)
- Modern processors use several techniques to avoid the speed penalty which occurs due to random external memory access.
- Most processors these days have an on-chip cache. “Local” read/write operations occur through these caches, which is much faster.
- Cache replenishment is carried out in a burst mode which is relatively fast and efficient.
- Instruction fetching is typically from sequential addresses and occurs through dedicated instruction caches.
- However, a jump to a far address may involve a speed penalty, making if-then-else kind of software constructs inefficient.

Load/store Architecture

- In a Von Neumann architecture (using a single bus), a data read or write operation from memory (as a part of instruction execution) can delay fetching of the next instruction.
- Modern processors therefore use a Load/Store architecture. The only instructions which access data memory are the ones for loading data from memory to a register or for storing register contents into memory.
- All other instructions operate only on internal registers.
- Thus the result update at the end of an instruction does not interfere with fetching (or pre-fetching) of instructions.

ARM: A pipelined Load/Store RISC processor

ARM is an acronym for Advanced RISC Machines. It is an unusual processor in may respects.

- Its designers do not manufacture the processor themselves, but licence its design to other manufacturers who incorporate the design in their own processors and other products.
- It is a 32 bit processor which supports two distinct instruction sets and allows switching from one instruction set to the other dynamically. The instruction sets are:
  1. ARM instruction set with 32 bit instructions, and
  2. THUMB instruction set which uses 16 bit instructions with 32 bit data size.
- More recent versions of ARM are available with a 64 bit architecture and which optionally support a third instruction set called Jazelle which uses an 8 bit instruction set and directly implements JAVA byte code.

ARM: Data Size

- Data registers in ARM are 32 bits wide.
- However, it can operate on operands of different size.
  - 8bit Bytes
  - 16bit Half Words
  - 32bit Words
- The memory is byte oriented and addresses are in units of bytes.
- Each word occupies 4 address locations. Their order can be defined as Big-Endian (MSB at lowest address) or as Little-Endian (LSB at lowest address).
- The Endian convention can be set using a configuration register.

ARM: Operation Modes

- The processor operates in one of seven “modes” at any given time.
- Modes are a generalisation of the interrupt mode in other processors.
- The processor enters a new mode when specific exceptions – like an external interrupt or internal exception occur.
- At power on, the processor begins operation in “Supervisor” mode. When a user program is loaded for execution, it runs in the “User” mode.
- The mode determines which registers are in active use and which instructions are permitted to be executed. (Some instructions are “privileged” and may be used only in privileged modes).
- For example, the operating system will use a privileged instruction to change the mode from supervisor to user when it loads and runs a user program.

ARM: Operation Modes

The ARM can operate in one of these seven basic modes:

Supervisor: It is entered on reset or when a Software Interrupt instruction is executed.

User: Unprivileged mode under which most tasks run.

FIQ: This mode is entered when a high priority (fast) interrupt is raised.

IRQ: is entered when a low priority (normal) interrupt is raised.

Abort: is used to handle memory access violations.

Undef: is used to handle undefined instructions.

System: This is a privileged mode which uses the same registers as the user mode.

New versions of ARM add an eigth mode called “Monitor” which is used for debugging. All modes except User are “privileged” modes.

ARM: Hardware resources – the register set

- ARM has 37 registers, each of which is 32 bits wide.
- Different physical registers may be bound to the same register name, depending on the mode in which the processor is currently executing.
- For example, there are five different physical registers which can be bound to the register name SPSR (saved program status register) depending on the mode.
- Every mode, except the user and system modes have a different physical register which is accessed when the program invokes the register name “SPSR”.
- The advantage of this scheme is that when an overlaid register is modified in a given mode, registers carrying the same name in other modes remain unchanged.
- Thus an overlaid register need not be saved/restored when it is used in a particular mode – such as in an interrupt service routine.

ARM register set

- An ARM program can use 16 registers (R0–R15) and flag registers called current program status register (CPSR) and saved program status register (SPSR). All these registers are 32 bits wide.
- Registers R0-R7, R15 and CPSR refer to the same physical registers in all modes.
- Different physical registers may be mapped to the same register name, depending on the current mode.
- On the other hand, different bit fields in the program status registers CPSR and SPSR may be accessed using different names.
  Thus APSR (application program status register) refers to the flag bits (and a few others) in CPSR. Other bit fields (CPSR bits 7:0) related to interrupts are referred to as ITSTATE register.

ARM Registers: Program Counter: R15

- R15 is the program counter (also referred to as PC) and contains the address of the next instruction to be fetched. Notice that due to pipelining, this is **not** the address of the instruction being currently executed.
- A write operation to the program counter is equivalent to a jump to a new address, and results in flushing of the instructions which have been pre-fetched (assuming sequential execution) but not yet executed.
- In ARM state, instructions are 32bits (4 bytes) in size. Therefore all instruction addresses are multiples of 4. Thus the two least significant bits of R15 must be zero.
- In THUMB state, instructions are 16bits (2 bytes) in size. Therefore all instruction addresses should be even. Thus least significant bit of R15 must be zero.
- In some ARM architectures, a change of instruction state can be requested by providing a destination address in a branch (BX) instruction with non-zero LSB values.
- All modes use the same physical register as PC or R15.

ARM Registers: Stack Pointer: R13

- Register R13 is also referred to as the stack pointer or SP.
- This register can be used just like any other register. However, its contents are used in a special way by some instructions to implement a stack.
- The stack pointer SP is implicitly used by load and store instructions in pre or post indexed addressing of memory operands.
- There are six different physical registers which are used as R13, depending on the current mode.
- User and System modes use the same register set. Other than this, every mode uses a different physical register as R13 or SP.
- This permits implementation of independent stacks in each mode.

ARM Registers: The Link Register: R14

- Many processors use the stack to store the return address during function calls and interrupt handling.
- However, the stack resides in memory and this can be quite inefficient due to slow memory access.
- ARM uses the register R14 to store the return address during function calls and interrupts. It is therefore also known as the link register or LR.
- The programmer is responsible for saving and restoring R14 in case of recursive calls.
- Since interrupts do not occur under program control, a dedicated separate physical register is provided for each mode, which serves as R14 in that mode. The service routine invoked to handle the interrupt uses the contents of R14 as the return address.
- Since R14 is a different physical register for each mode, the return address is properly preserved even for successive interrupts.

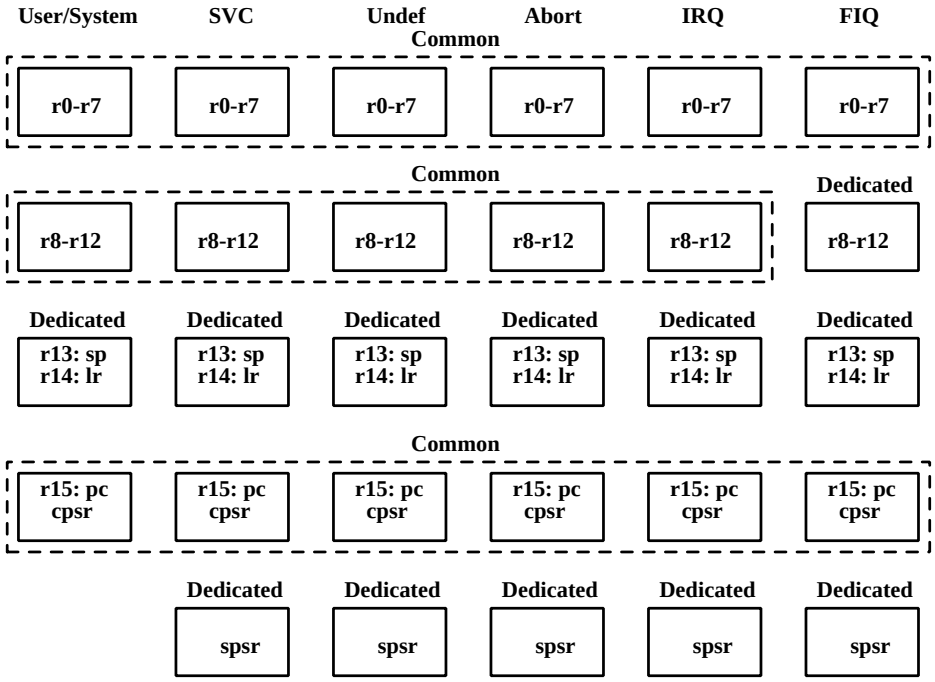
ARM registers: The Saved Program Status Register

Program status registers are a generalized version of the flag and interrupt status registers used in other processors.

- When an interrupt occurs, the flag register in addition to any used registers must be saved by the interrupt service routine.
- In case of ARM, this happens automatically and the current program status register is copied to the saved program status register of the destination mode, so that the program status can be restored when the interrupt service routine returns.
- Each mode (except User and System modes) has its own dedicated physical register for saving a copy of the current program status register. This register is accessed as SPSR in all modes.

Register Overlay Scheme for ARM

The figure below shows the register overlay scheme of ARM.



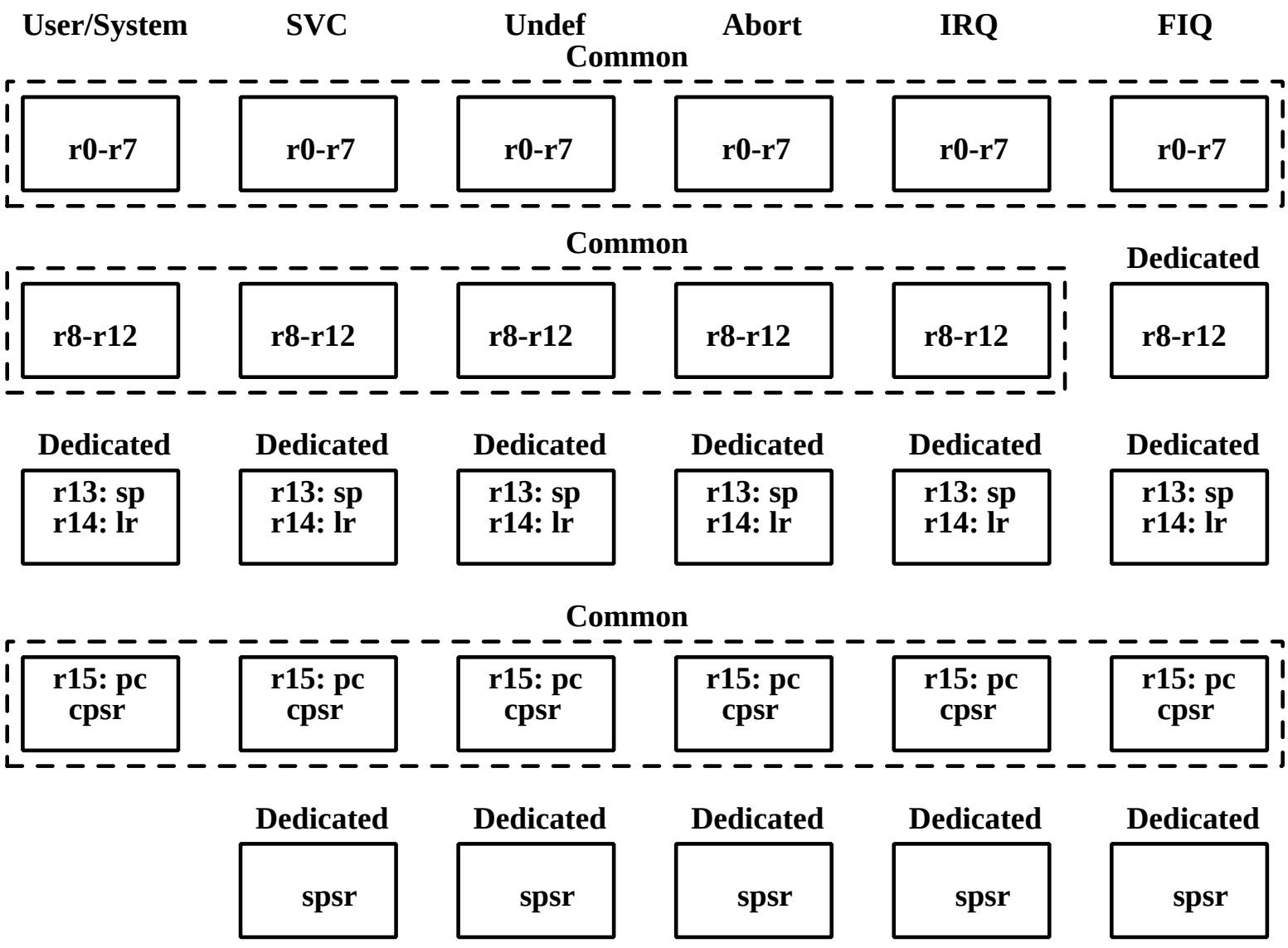
Notice that the FIQ mode provides dedicated data registers R8-12. These registers can be used in the interrupt service routine without having to save them.

Common registers R0-R7 can be used to store information which the service routine can access in read only mode.

All modes except User and System have dedicated registers for R13 (SP), R14 (LR) and SPSR.

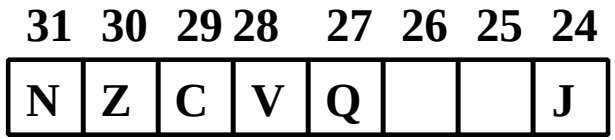
Program Counter (R15) and the current program status register (CPSR) are common to all modes, of course.

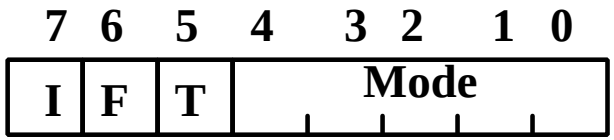
Register Overlay Scheme for ARM



The program status registers

The top byte contains the traditional condition codes.
The bottom byte carries interrupt masking and mode information.





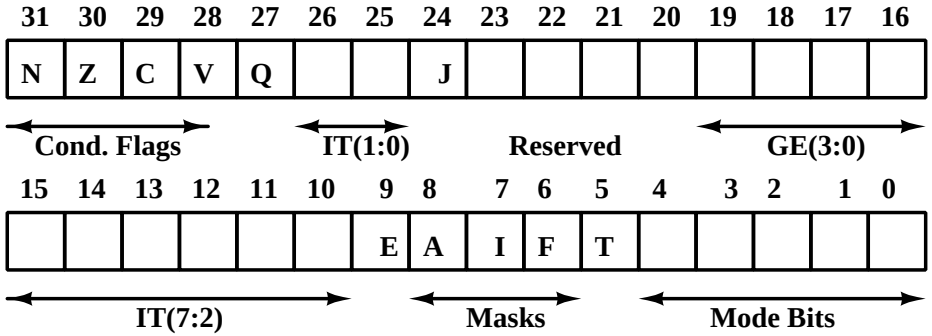
N: Negative result, Z: Zero result, C: Carry, V: Overflow.
Q bit is set if saturation occurred during addition.
J bit indicates that the processor is currently executing the Jazelle instruction set.

Bit 7 is set to disable (mask) IRQ interrupts.
Bit 6 is set to mask FIQ (fast) interrupts.
Bit 5 indicates that THUMB instruction set is being executed.
Bits 4-0 encode the current mode.

Q and J bits are meaningful only in some ARM architectures.
Q bit is used if the processor provides saturating addition.
J bit is used if the processor supports the 8 bit Jazelle instruction set.

Updated program status registers: APSR

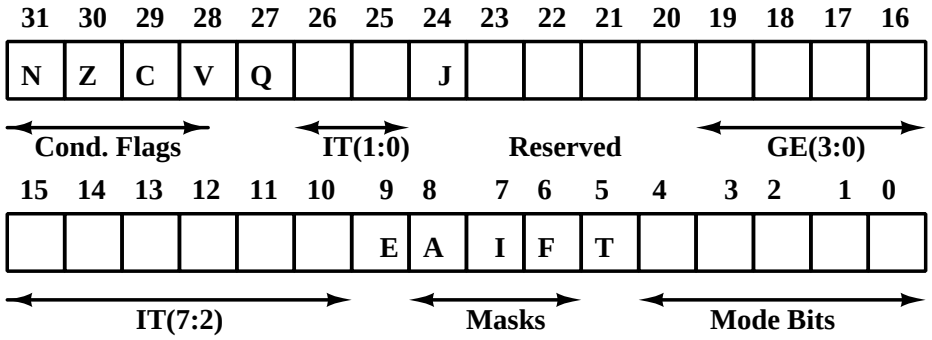
The use of program status register has been updated in the recent ARM-cortex processors, and more bits have been defined.
Combinations of selected bit fields in the program status register are now called by different register names.



APSR or Application Program Status register accesses bits 31-27 and bits 19-16.

- Bits 31-27 retain the functions assigned to them as described for CPSR (condition flags).
- Bits 19-16 contain GE (Greater than of Equal to) flags.

Updated program status registers: APSR



A new SEL instruction has been added to the THUMB instruction set, which can select bytes from either of two source operands depending on the value of GE bits.

SEL R_d, R_{s1}, R_{s0}

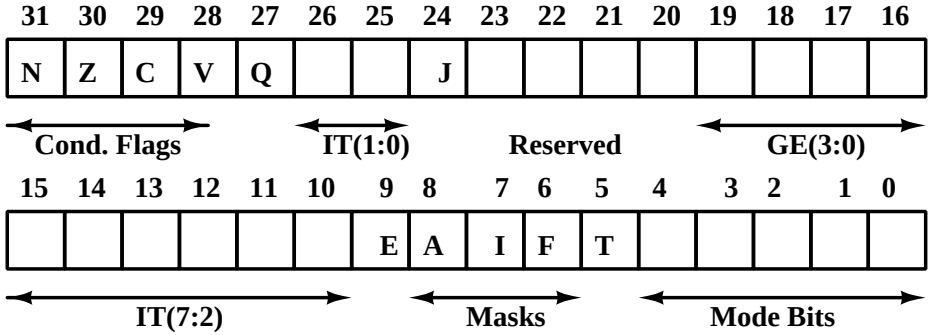
The 4 GE bits (bits 19:16 of CPSR) decide (for each byte in the destination register), whether the corresponding byte from the first or the second source register will be copied to it.

The GE bits are set by comparison instructions.

Thus SEL is like a selected byte-wise MOV in parallel, where the selection is based on a previous comparison.

Updated program status registers: Execution State Registers

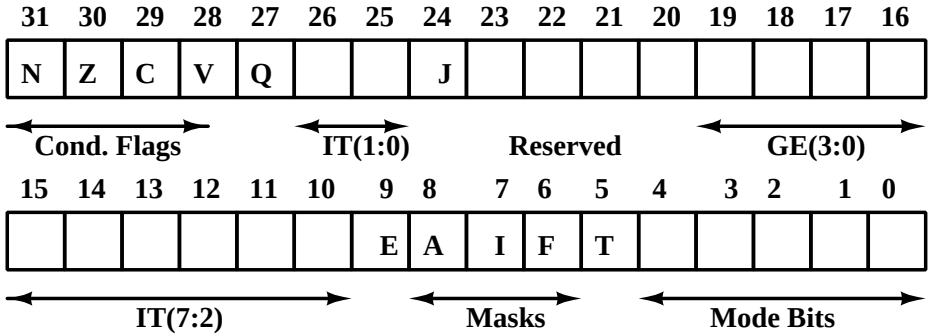
Bits J and T have the same function, but can be accessed as the instruction set state register.



Some other bits have been grouped and re-named

- E bit (bit 9) determines whether the data will be interpreted as Big-Endian (MSB at lower address) or Little-Endian (LSB at lower address).
- A bit (bit 8) disables asynchronous aborts.
  An asynchronous data abort occurs when the cache or memory system generates an error after it has signalled data transfer completion.
- I and F bits mask (disable) IRQ and FIQ interrupts, as described earlier.

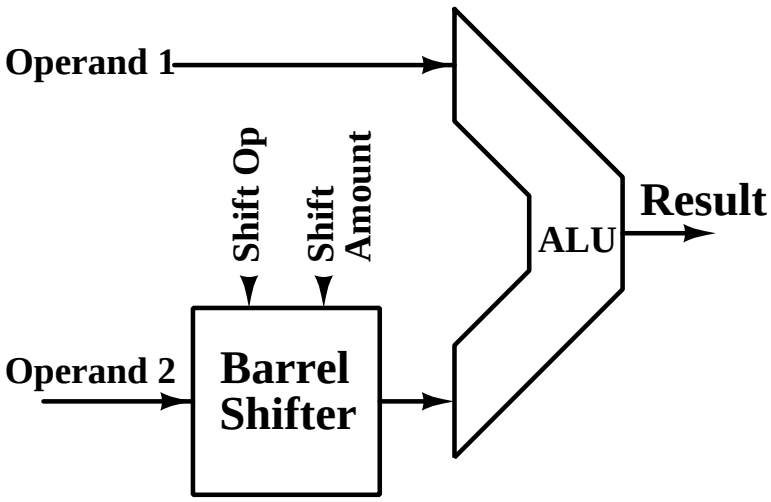
Updated program status registers: Execution State Registers



- The IT (if-then) instruction in THUMB instruction set permits conditional execution of up to 4 following instruction in THUMB state.
- The IT bits (CPSR bits 15:10 and 26:25) are used for resuming operation after an interrupt in the middle of an “IT Block”.
- (Earlier, conditional execution could only be used in ARM state, THUMB state did not permit conditional execution).

Barrel Shifter for Second Operand

Apart from the register set, ARM provides a barrel shifter for pre-processing an operand. The barrel shifter operates on the last operand of a data processing instruction.



In a data processing instruction, the second operand can be modified through a barrel shifter and/or inverted before the instruction operates on it.

12 of the 32 bits in the instruction are used for specifying the shift/rotate operation.

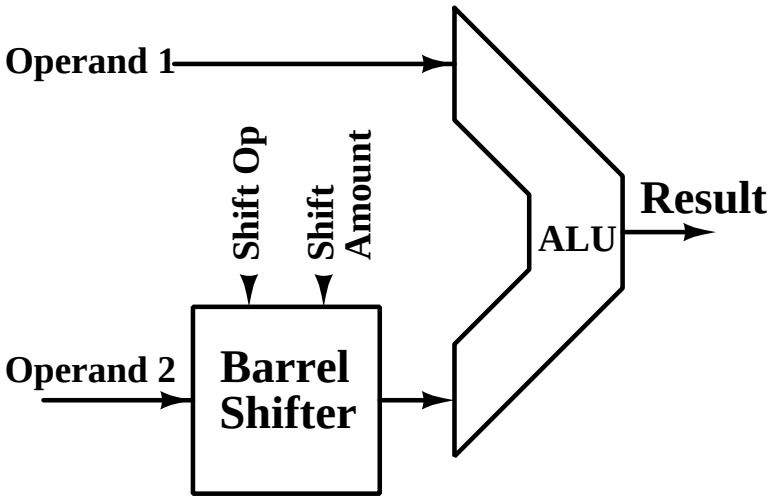
The barrel shifter provides the following operations:

- LSL a logical left shift, 0's inserted from the right.
- LSR a logical right shift,0's inserted from the left.
- ASR arithmetic right shift: msb re-inserted from the left.
- ROR rotate right.
- RRX rotate right with carry.

Barrel Shifter for Second Operand

- If the second operand is a register, the shift value can be an unsigned 5 bit integer (0 to 31).
- The shift value can also be given in a register, enabling a shift operation by a variable number of positions.
- The original operand remains unchanged. The modified value is used Only for the data processing instruction.
- This is useful for scaling a value before operating on it.

Barrel Shifter for Second Operand



If the second operand is an immediate value, our choices are rather limited.

This is because the immediate value, as well as the shift amount have to be accommodated in the 12 bits available for specifying the shift operation.

- The immediate value is supplied as an 8 bit quantity.
- Only right rotate operation is allowed.
- The amount by which a right rotation is carried out must be even. This is because we have only 4 bits left to specify the rotation amount.

Exception Handling

Exceptions result in a change in the mode of operation of the ARM processor.
The following types of exceptions are recognized:

Exception type	Destination Mode
Reset	Supervisor
Undefined Instruction	Undefined
Supervisor Call	Supervisor
Pre-fetch Abort	Abort
Data Abort	Abort
IRQ interrupt	IRQ
FIQ interrupt	FIQ

Exception Handling

When an exception occurs:

1. Hardware determines the destination mode for the exception.
2. The return address is saved in the link register of the destination mode.
3. The current program status register value (CPSR) is copied to the saved program status register (SPSR) of the destination mode.
4. CPSR bits are adjusted as appropriate for the new context.
5. Exception vector corresponding to the destination mode is loaded into PC. Since a new value is loaded into PC, the pipeline is flushed.
6. The processor starts executing from the address held in PC.

Exception Handling: CPSR update

The CPSR register is updated as follows (after saving a copy in SPSR of the destination mode):

- The mode bits are set to the destination mode.
- Interrupts for priority level lower or equal to the current exception are masked. For example, when FIQ occurs, FIQ as well as IRQ is masked by setting the F and I bits.
- Instruction set state is set to a pre-determined value obtained from a configuration register: most commonly to ARM.
- Endian value is set to a pre-determined value obtained from a configuration register.
- The IT bits are cleared.

Return from an exception

When the service routine for an exception ends, control returns to the interrupted code through the following two actions:

1. SPSR value is copied back to CPSR.
2. The return value stored in LR is copied back to PC.

Notice that restoring CPSR value restores not only the condition flags but also the instruction state (ARM or THUMB), the original interrupt flags and the status of IT flags which track conditional execution of multiple instruction in the THUMB state.
For current versions of ARM cortex processors, it is recommended to use a branch instruction:
BX LR
rather than a MOV instruction:
MOV PC, LR
Since either of these will load a new value into PC, the pipeline will be flushed.

Conditional Execution of Instructions

- Most processors provide a conditional branch instruction.
- In the ARM instruction set, nearly *all* the instructions can be executed conditionally.
- Instructions carry a condition field in their encoding.
- The instruction is executed only if the condition is consistent with the current value of flag registers. Otherwise it is treated as a NOP.
- This results in efficient code generation.

Conditional Execution of Instructions

suffix	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	Plus 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	$N \neq V$
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or $N \neq V$
AL	Always	

Conditional Execution of Instructions

- ARM instructions can be made to execute conditionally by post-fixing the instruction mnemonic with the appropriate condition code.
- If no condition code is specified, it is assumed to be “Always”.
- Data processing instruction do not set the condition codes by default. To set condition codes, the mnemonic should be suffixed with an ‘S’. This permits preserving condition codes across multiple data processing instructions if needed.
- The purpose of the CMP instruction is to set condition codes based on comparison. So this instruction does not need an ‘S’ to set condition codes.

Conditional Execution of Instructions

Consider the case when we want to add R1 to R2 and store the result in R0 only if R3 is $\neq 0$.

The traditional code for this would have been:

```
      CMP   R3, #0
      BEQ   L1
      ADD   R0, R1, R2
L1:   ...
```

Using conditional execution, this can be coded as:

```
CMP     R3, #0
ADDNE   R0, R1, R2
...
```

Thus conditional execution of instructions improves code density and performance by reducing the number of forward branches.

Conditional Execution of Instructions

Conditional codes can be used in various forms and contexts:

- Using a sequence of multiple conditional instructions:
  Consider the implementation of pseudo code:
  if(a == 0) func(27);

```
CMP     R0, #0
MOVEQ   R0, #27;   set the argument
BLEQ    func;      call func: Branch and Link if equ
```

- Set flags then use different conditions
  Consider the implementation of pseudo code:
  if (a == 0) x=0;
  if (a $>$ 0) x=1;

```
CMP     R0, #0;    Set condition flags
MOVEQ   R1, #0;    Value if EQ condition is true
MOVGT   R1, #1;    Value if GT condition is true
```

Conditional Execution of Instructions

We can Execute the compare instructions themselves conditionally.

Consider the implementation of pseudo code:
if (a == 4 || a == 10) x=0;

```
CMP     R0, #4;     Set condition codes
CMPNE   R0, #10;    executed only if R0 ≠ 4
MOVEQ   R1, #0;     executed if R0 = 4, else if R0=10
```

The first instruction sets the Z flag only if $R0 = 4$.
If $R0 = 4$, the second comparison is not carried out at all (NE condition) and we go to the third instruction, which stores 0 in R1 because Z flag is set.
If $R0 \neq 4$, the second comparison will be carried out and will set the Z flag if $R0 = 10$.
Now the third instruction will store 0 in R1 if the value of R0 was 10.

Data Processing Instructions

Syntax: <Operation>{<cond>}{S} Rd, Rn, Op2
Op2 is routed through the barrel shifter.

- Data movement instructions do not specify Rn.
- Comparison statements do no specify Rd.
  (These only set flags – S need not be specified).

Data Movement: MOV, MVN – src op is inverted by MVN

Arithmetic: ADD, ADC, SUB, SUBC, RSB, RSC

RSB and RSC perform reverse subtraction *i.e.* subtract Rn from Op2. – (Why are these required? couldn't we just switch the operands of SUB or SUBC?)

These instructions are useful because barrel shifter ops are available only for op2. Given x and y, direct computation of expressions like 8x -y cannot be done with just SUB and SUBC.

Data Processing Instructions

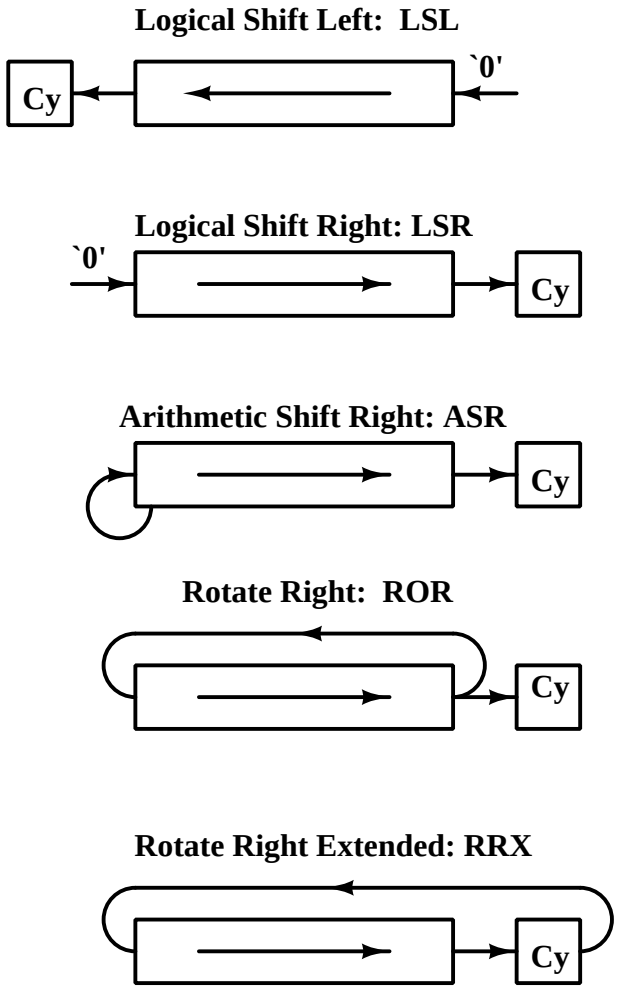
Syntax: <Operation>{<cond>}{S} Rd, Rn, Op2
Op2 is routed through the barrel shifter.

Logical: AND, ORR, EOR, BIC
AND, ORR and EOR compute the AND, OR and XOR functions.
BIC clears those bits of Rn where Op2 has a 1.
(Rd = Rn AND $\overline{\text{Op2}}$)

Comparisons: CMP, CMN, TST, TEQ
CMN compares with complement of Op2.
TST performs an AND of Rn with Op2 while TEQ performs EOR of Rn with Op2. Both of these then discard the result. The computation is used only for setting flags, as in the case of CMP and CMN.

Use of Barrel Shifter

Syntax: <Operation>{<cond>}{S} Rd, Rn, Op2
Op2 is routed through the barrel shifter.



- If Op2 is derived from a register, we can specify LSL, LSR, ASR, ROR or RRX as an operation to be performed on the register, before the value is used by the data processing instruction.
- If Op2 is derived from an immediate value, the only choice is ROR. The immediate value can only by 8 bits wide and the rotation is by an even number of places.

For example, MOV R0, 0x40 ROR 26 will load the constant 4096 into R0. (ROR 26 effectively rotates by 6 positions to the left).

Use of Barrel Shifter

Examples of the use of Barrel Shifter:

MOV R2, R0, LSL #2

$\Rightarrow$ Shift R0 left by 2, write to R2, (R2=R0x4)

ADD R9, R5, R5, LSL #3

$\Rightarrow$ R9 = R5 + R5 x 8 or R9 = R5 x 9

RSB R9, R5, R5, LSL #3

$\Rightarrow$ R9 = R5 x 8 - R5 or R9 = R5 x 7

SUB R10, R9, R8, LSR #4

$\Rightarrow$ R10 = R9 - R8 / 16

MOV R12, R4, ROR R3

$\Rightarrow$ R12 = R4 rotated right by value of R3

Multiply operations

For MUL and MULA, the product size is truncated to 32bits (32 least significant bits are retained!):

MUL{<cond>}{S} Rd, Rm, Rs
$\Rightarrow$ Rd = Rm * Rs
MLA{<cond>}{S} Rd, Rm, Rs, Rn
$\Rightarrow$ Rd = (Rm * Rs) + Rn

MLA is used for multiply and accumulate operations.

Overflow flag is set if the product exceeds 32 bits.
These instructions retain only the 32 least significant bits of the result. Therefore these are suitable only for unsigned multiplication of relatively small numbers.

Multiply operations

Multiplication with Product size = 64bits: (long multiply)

{U}{S}MULL{<cond>}{S} RdLo, RdHi, Rm, Rs
$\Rightarrow$ RdHi, RdLo = Rm * Rs
{U}{S}MLAL{<cond>}{S} RdLo, RdHi, Rm, Rs
$\Rightarrow$ RdHi, RdLo = (Rm * Rs) + RdHi, RdLo

U is used for unsigned multiplication, S for signed.

There is no inbuilt divide instruction in ARM v7 architecture.

Generating Immediate Constants

- Instruction size is fixed (= 32 bits) and immediate constants must fit within the instruction.
- This means a full width (32 bit) constant cannot be moved into a register using the MOV instruction with an immediate operand.
- However, many long constants can be generated by combining barrel shifter operations optionally with negation through the use of instruction MVN.

The shifter rule for immediate operands is:
8 bit constant rotated right by an even number of places

A few examples:
ADD R1, R2, 0xFF ROR 16 will add 0xFF0000 to R2 and put the result in R1.
MVN R0, #0 will move 0xFFFFFFFF into R0

Generating Immediate Constants

- Normally we just specify the desired constant in the assembly code and the assembler tries to translate it to the appropriate construct.
  Thus, MOV R0, #4096 will be assembled as
  MOV R0, 0x40 ROR 26 by the assembler.
- Not all constants can be so constructed. If it is not possible to find a construct which will synthesize the desired constant, the assembler will issue an error.
- In such cases, the constant must be loaded from memory. (The LDR instruction loads data from memory into a register).
- LDR is, however, much slower than MOV, and a MOV instruction is preferred it it is possible to construct the immediate value using it.

Loading Immediate Constants: Literal Pools

- ARM assemblers provide a pseudo-instruction using LDR, but with an equal to ('=') sign before the constant value to choose the fastest way for loading constants.
- The assembler replaces it with a MOV if possible. Otherwise it uses LDR (without the =) and loads the constant from memory.
- For example, LDR R0,=0xFF assembles to MOV R0,#0xFF, because it is possible to generate the constant using MOV.
- However, LDR R0,=0x55555555 will assemble to
  LDR R0, [PC,#Imm12]
  where the constant 0x55555555 is stored in memory at the offset #Imm12 relative to the address that PC is pointing to.
- The block in memory where such constants are placed is called a literal pool.
- Literal pools are typically placed between the end of a function and the beginning of the next, so that these can be accesses with a 12 bit offset from the current value of PC.

Generating Pointer Values

- A particular case of interest is for loading pointers into a register.
- A pseudo instruction ADR loads the address of a given location in a register.
- The assembler constructs the value in the register by calculating the offset from PC and using the add or sub instruction to generate the pointer value.

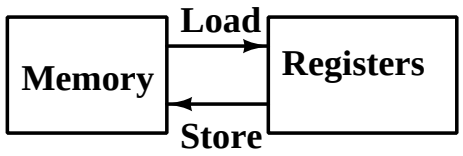
Example

```
start   MOV   R0,#10
        ADR   R4,start;   assembles to SUB R4,PC,#0xc
```

Because of pipelining, PC is pointing to 8 locations ahead of the statement replacing ADR.
The label “start” is in the previous line
(4 bytes before the statement).
So the value of the pointer is computed by subtracting 12 (decimal) from PC.

Load and Store Instructions

ARM has a load/store architecture. Therefore no operations are carried out directly on memory operands.



The only instructions which access the memory are those for loading and storing to/from registers.

Load/Store operations can be carried out for words/half-words or bytes.

LDR	STR	word sized (32 bit) transfer
LDRB	STRB	Byte transfer
LDRH	STRH	Half word (16 bit) transfer
LDRSB	Loading a signed byte	
LDRSH	Loading a signed half word	

Syntax:

LDR/STR {<cond>}{<size>} Rd, <address>

The address is specified by a base register with an offset.

Address specification in Load/Store Instructions

- For word and unsigned byte accesses, the offset value can be An unsigned 12 bit immediate value or a register optionally shifted by an immediate value.
- This offset can be added or subtracted from the base register.

Examples:

LDR R0, [R1,#8]; STR R0,[R1,-R2];
LDR R0,[R1,-R2,LSL#2]

- For half-words and signed half-words or bytes, the offset value can be an unsigned 8 bit constant or an unshifted register.

Pre-indexing in Load/Store Instructions

- Pre-indexing means the address is calculated before the load/store operation and the computed address is used for data transfer.
- The computed value can be optionally used for updating the base register.

Assume that R0 contains 0x5 and R1 contains 0x200. Then,

STR R0, [R1, #12] will *first* compute R1 + 12 (decimal) = 0x20C.

It will *then* store R0 at this address.

Thus the word at location 0x20C will contain 5 after the operation is complete.

We can optionally follow the address spec with a "!".
STR R0, [R1, #12]! Here the computed value for the address (0x20C) will update the value in R1.

Post-indexing in Load/Store Instructions

In post indexing, the address is computed *after* the load/store operation.
The only use for this is to update the base pointer register after the load/store operation.

- To indicate post indexing, we place the offset value outside the square brackets in the address specification.
- STR R0, [R1], #12 will store the value of R0 (=5) at the *current* value of R1 – that is at 0x200 and *then* compute 0x200 + 0x0C = 0x20C.
- The only use of this computation is to update the pointer, so the bang sign is not required.
- R1 is updated to this computed value (0x20C).

pre and post indexing with update can be quite useful when LDR/STR is used repeatedly.

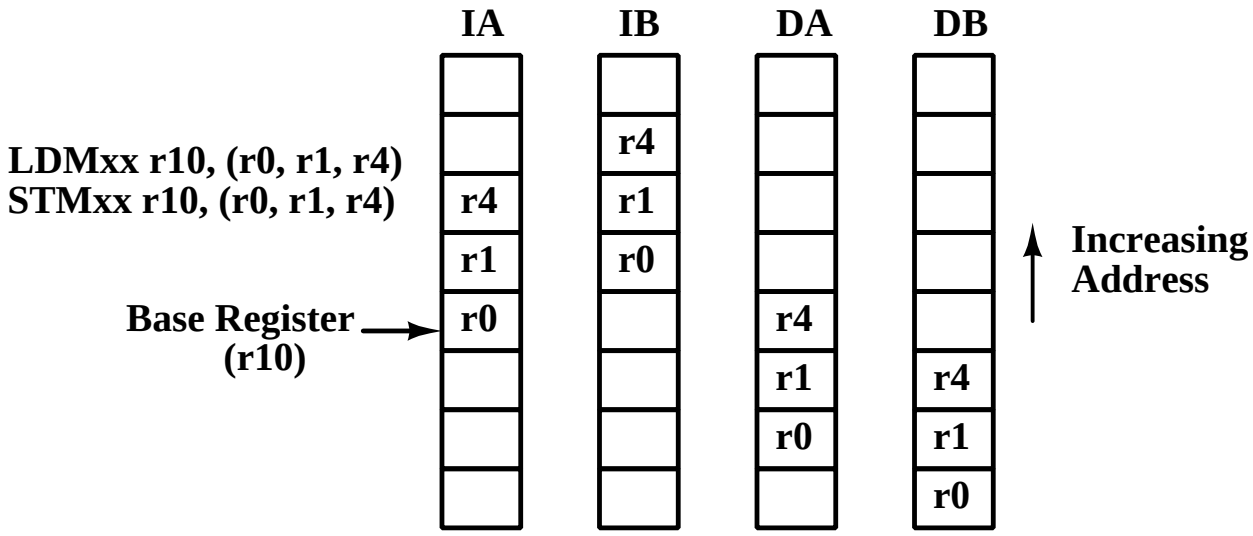
Load/Store for multiple registers

- Instructions LDM and STM can load or store multiple registers.
- Loading or storing a list of registers in a single instruction will take several machine cycles. (This is a departure from pure RISC operation, of course!)
- The list of registers is implemented as a bit field. Therefore the order of registers in the list is immaterial.
- These instructions carry a pointer which is incremented/decremented with every load and store operation.
- The direction of pointer change and the order of load/store and pointer update is indicated by suffixes following LDM and STM.

Load/Store for multiple registers

Syntax: for LDM and STM instructions:

<LDM|STM>{<cond>}<addressing mode> Rb {!}<register list>



The addressing mode can be:
IA (increment after),
IB (increment before),
DA (decrement after)
or DB (decrement before).

- Order of specification in the register list is immaterial.
- Registers are always stored such that the higher index register is at a higher address.
- The optional bang sign (!) updates the pointer with its final value if it is present.

LDM/STM at the entry/exit of functions

- Apart from saving/restoring all used registers at the entry/exit of functions and exception handlers, this construct can be used for saving the return address and executing the return.
- This can be done by including R14 (Link Register) in the saved register list and replacing LR by PC in the restored register list.

For example, let us take the case when registers R1, R9 and R12 are to be preserved by a called function. On entry to the function we execute
STMDB SP!, {R1, R9, R12, LR}
While returning, we execute: LDMIA SP!, {R1, R9, R12, PC}
This will restore R1, R9 and R12 and will also copy the saved return address in LR to PC. This effectively performs the return.

Status Register Access instructions

- We cannot read or write to the status registers CPSR and SPSR with MOV instructions.
- Instructions MRS and MSR copy data from and to a status register.
- Instruction MRS is used to move the value of a status register to a general purpose register.
- MSR moves the contents of a general purpose register to a status register.
- In User mode, only the condition flags may be accessed and altered. A privileged mode is required to alter other bits.

Branch and Call processes

- The branch instruction executes a jump to a given label. The syntax is:
  B{<cond>} Label
- The instruction actually carries an offset of the label from the current value of PC. The processor adds this value to PC to perform the jump.
- Since the two least significant bits of an address are always 0 in ARM instructions, these are not stored in the instruction.
- The stored value is 24bits wide. The processor appends '00' to it, sign extends it to 32 bits and adds it to PC.
- The pipeline is flushed because a write has been performed on PC, and the next instruction will be fetched from the updated value in PC.

Function Calls in ARM

- Instruction BL acts similarly to the Branch instruction B, but it also stores the return address in the link register before changing the value of PC.
- The syntax is: BL{<cond>} Label
- This effectively performs a function call. The called function (starting from Label) should restore PC from the link register to perform a return.
- If the called function calls yet another function, this would overwrite the return address in LR. Therefore the called function should save and restore the link register if it is going to call another function.

Function Call and return with Register Save

- Called functions are often required to save and restore data registers.
- One can combine register save/restore with function return using STM and LDM.
- For example, take a function which needs to save and restore registers R0, R1 and R4.
- On entry to the function, it executes:
  STMDB SP!, {R0, R1, R4, LR}
  When returning, it should execute:
  LDMIA SP!, {R0, R1, R4, PC}
- This saves and restores R0, R1 and R4. Additionally, it saves LR and restores the saved value to PC, effectively loading the saved return address into PC and thus performing a return.

Branch to a Variable Address

- The BX Instruction works similarly to B except that the destination address is given as a register and not as a label.
  BX{<cond>} Rn
- Thus, the instruction “BX LR” will jump to the address stored in the Link register, effectively performing a function return.
  If the return is not combined with multiple register save and restore, BX LR is the recommended way to return from a function.
- BX can also be used for changing state to ARM or THUMB instruction set.
  The used destination address will always have the least two significant bits of the address as 0. However, depending on the least significant bit stored in Rn, the state is changed to ARM (if this bit is 0) or to THUMB (if the bit is 1).

Branch and Link to a Variable Address

- Instruction BLX work similarly to BL. The destination address may be given as a register or a label.
  BLX{<cond>} Rn BLX{<cond>} label
- BLX can be optionally used with a label rather than a register as its argument.
- When used with a label, it always toggles the current instruction set state from ARM to THUMB and from THUMB to ARM.

Returning from an exception

- Returning from an exception requires that PC should be restored from LR as also CPSR should be restored from SPSR.
- If LR is moved to PC without first restoring CPSR, the next instruction will be from the main program and we cannot restore CPSR from SPSR!
- If we move LR to PC *after* restoring CPSR, the mode would have changed and LR would no more refer to the LR of the exception mode which contains the return address. So we cannot return!!
- Therefore the two opeartions need to performed in a single instruction. Special constructs are therefore used for returning from an exception.
- We have two different ways of returning from an exception depending on whether registers need to be saved/restored or not.

Returning from an exception

- If registers need to be saved/restored, we use STM and LDM instructions.
- For example, assume that a full descending stack is being used and all registers have to be saved: on entry to the exception handler we do
  STMDB SP! {R0-R12, LR}
- While returning from the exception, we use LDM to restore registers and add a hat symbol to it to simultaneously restore CPSR from SPSR. Thus:
  LDMIA SP!, {R0-R12, PC}^ will restore PC from LR and simultaneously restore CPSR from SPSR.
- If no registers are to be restored by the handler, we use a flag setting artithmetic instruction for loading the return address into PC. Thus:
  MOVS PC, LR moves LR to PC and also restores CPSR from SPSR. (One could also use SUBS PC, LR, #0).

SWP and SWPB

ARM provides atomic instructions SWP and SWPB, which exchange a word or a byte between a register and a memory location.
The syntax is:
SWP{<cond>} Rd, Rm, [Rn]
SWPB{<cond>} Rd, Rm, [Rn]

- The instruction reads a word (byte in case of SWPB) from the address [Rn] and places it in register Rd. It takes the value in Rm, and places it in memory at address [Rn].
- Because the two actions are in the same instruction, these are indivisible or atomic. This avoids some critical code hazards.
- If Rd and Rm are the same register, this swaps the values between Rd and the location [Rn] – hence the name.

This can be used to remove the critical code hazard case discussed earlier.

Semaphore Instructions

Assume that we have a shared resource, which we can mark as busy or free using a byte flag stored in memory at the address stored in R1.
The flag value is 00 if the resource is free and 0xFF if it is busy.
A critical hazard exists if the process is interrupted between checking the value of this flag and marking it as busy.

We pre-load R0 with 0xFF and execute: SWPB R0, R0, [R1]

- This puts the current value of the flag in R0 and *simultaneously* marks the flag as busy.
- We now examine R0. If the original value of the flag (now in R0) was “busy”, no change has been made in its value by writing to it, and we must suspend our process for now and try later.
- If R0 indicates “free”, we have already marked it as busy, so we can proceed to use the resource.

Semaphore Instructions LDREX and STREX

The SWP instructions are being replaced by instructions LDREX and STREX. LDREX and STREX reduce interrupt latency and can be used in a multi-processor environment.

- LDREX works like LDR, but it also “secures” the specific location from which LDR was executed for the exclusive use of the processor which issued LDREX.
- STREX works like STR. It also releases the lock on the memory location which was secured by LDREX.

Now a load and store will execute atomically for a given processor.
LDREX and STREX break up the SWP operation in two shorter steps. Also, the block access only to a given address, not to the entire bus and address space.

Software Interrupt using SWI

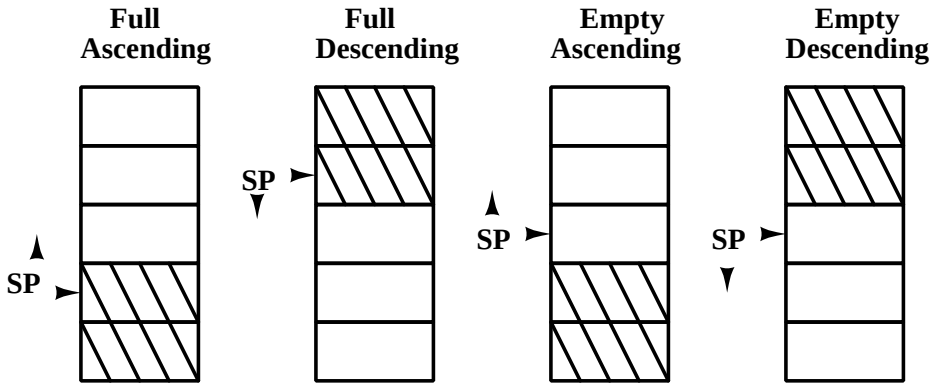
SWI{<cond>} SWI-number
The instruction is used for requesting a service from the operating system.

- By using the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.
- It causes an exception trap to the SWI hardware vector.
- The SWI handler can examine the SWI number to decide what operation has been requested.
- Based on the requested service, a branch is taken to the relevant subroutine, which executes in supervisor mode.
- A return from the exception will restore the save psr, which will set the mode back to User mode.

Implementing Stacks in ARM based systems

- Memory used by a process includes a “heap” and a “stack”.
- Heap typically contains variables and arrays for the running program.
- Stack is used for return addresses and local variables.
- Typically, the starts of heap and stack areas are initialized to opposite ends of the available memory and they grow towards each other.
- For example, for 8085, 8086, Pentium etc. the stack grows towards lower addresses.
- The stack for an 8051 grows upwards.
- Through its auto indexing system and load/store multiple mechanisms, ARM can implement all types of stacks easily.

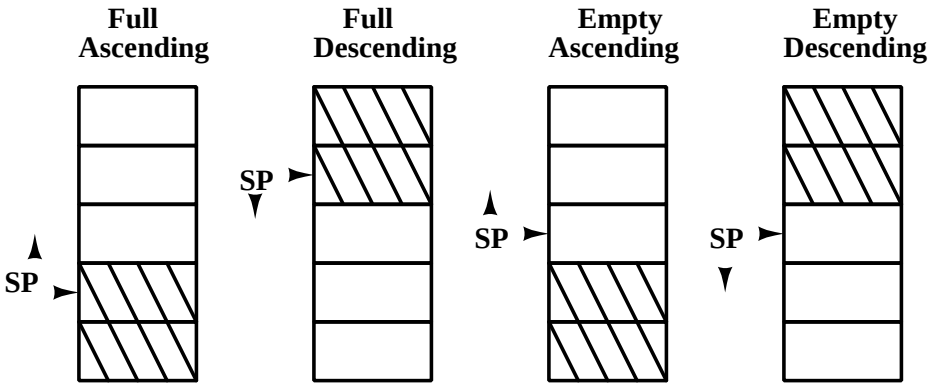
Full, Empty, Ascending and Descending Stacks



- A full stack is one where the stack pointer is pointing to the last item placed on the stack.
- An empty stack is one where the stack pointer points to the first available location on the stack.

- Stack pointer should be incremented when a new object is placed on an ascending stack.
- Stack pointer should be decremented when a new object is placed on a descending stack.

Full, Empty, Ascending and Descending Stacks

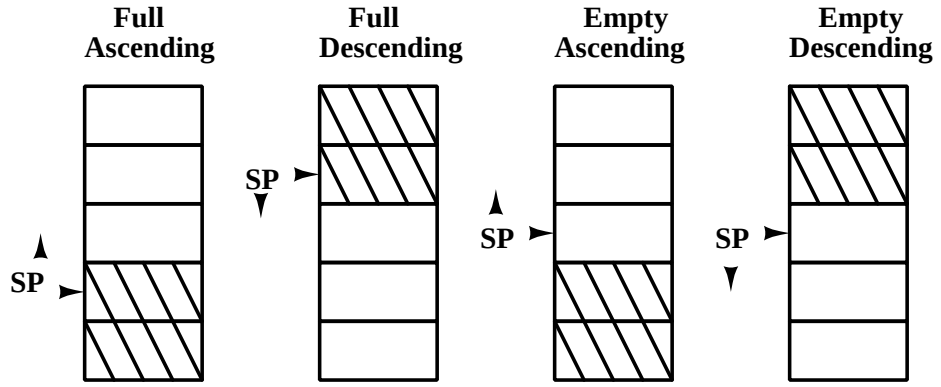


- In a full stack, the stack pointer must be adjusted *before* placing an new object on the stack, and *after* removing an object from the stack.
- In an empty stack, the stack pointer must be adjusted *after* placing an new object on the stack, and *before* removing an object from the stack.

Full, Empty, Ascending and Descending Stacks

- In a Full Ascending stack, the stack pointer is pre-incremented for a PUSH operation and post decremented for a POP operation. (This is the stack type used in 8051 systems).
- In a Full Descending stack, the stack pointer is pre-decremented for a PUSH operation and post-incremented for a POP operation. (This is the stack type in 8085, 8086, Pentium . . . systems).
- In an Empty Ascending stack, the stack pointer is post incremented for a PUSH operation and pre-decremented for a POP operation.
- In an Empty Descending stack, the stack pointer is post decremented for a PUSH operation and pre-incremented for a POP operation.

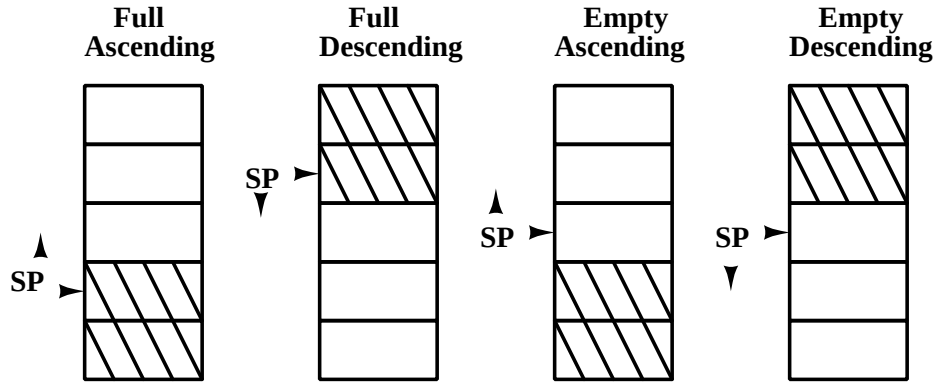
PUSH and POP operation for a single register in ARM



For a PUSH or POP of a single register, we can use:

- STR Rd, [SP + #4] ! for a PUSH and LDR Rd, [SP], -#4 for a POP in a Full Ascending stack.
- STR Rd, [SP - #4] ! for a PUSH and LDR Rd, [SP], #4 for a POP in a Full Descending stack.
- STR Rd, [SP] #4 for a PUSH and LDR Rd, [SP-#4] ! for a POP in an Empty Ascending stack.
- STR Rd, [SP] -#4 for a PUSH and LDR Rd, [SP+#4] ! for a POP in an Empty Descending stack.

PUSH and POP operation for multiple registers



For a PUSH or POP of a list of registers, we can use:

- STMIB SP!, {Reg. list} for a PUSH and LDMDA SP!, {Reg. list} for a POP in a Full Ascending stack.
- STMDB SP!, {Reg. list} for a PUSH and LDMIA SP!, {Reg. list} for a POP in a Full Descending stack.
- STMIA SP!, {Reg. list} for a PUSH and LDRDB SP!, {Reg. list} for a POP in an Empty Ascending stack.
- STMDA SP!, {Reg. list} for a PUSH and LDRIB SP!, {Reg. list} for a POP in an Empty Descending stack.

PUSH and POP operation for multiple registers

- The type of stack is generally decided once for all. PUSH and POP operations then use this kind of stack throughout in a program.
- Different suffixes must be used for PUSH and POP operations for a given type of stack. For example, for a Full Descending stack, one should use STMDB for PUSH and LDMIA for POP. This can be error prone.
- To simplify this, ARM assemblers accept stack type as the suffix rather than IB, IA, DB or DA.
- Thus, for a full descending stack, one can use STMFD and LDMFD for PUSH and POP operations.