# Lecture 4

# Subroutines

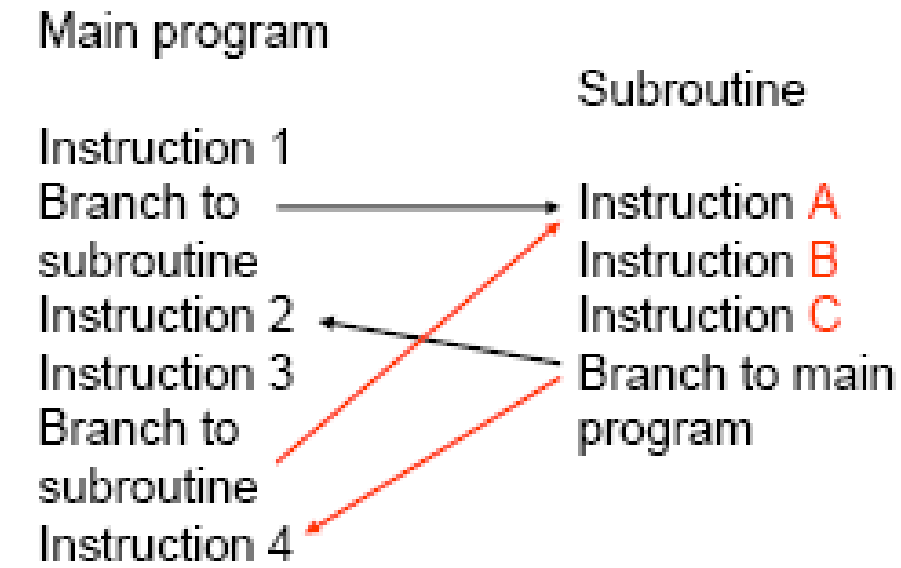
- Many microprocessor programs include groups of instructions which are repeated many times.
- It is wasteful of memory to include these instructions in the program again and again.
- Instead they can be included once in a special structure known as a subroutine.
- The main program branches to the start of the subroutine when it requires those particular instructions.
- At the end of the subroutine there is another branch back to the main program.

# Subroutines

So instead of this:

Instruction 1
Instruction A
Instruction B
Instruction C
Instruction 2
Instruction 3
Instruction A
Instruction B
Instruction C
Instruction 4

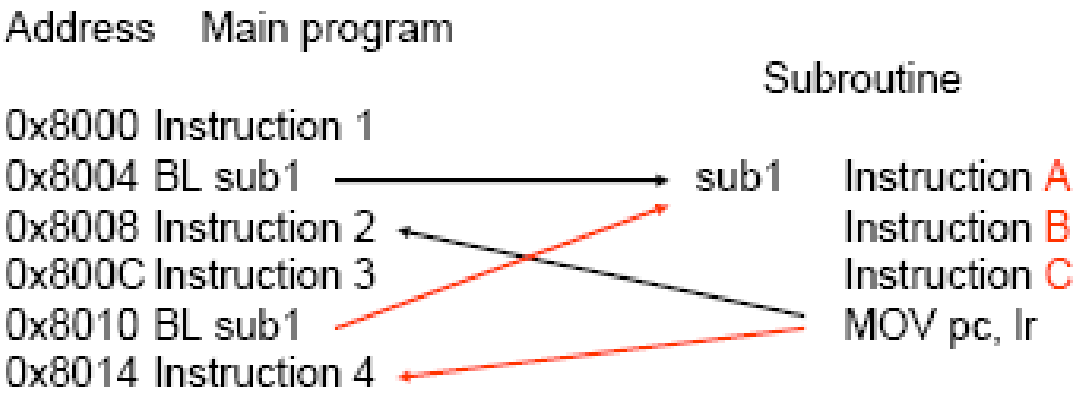
We have this:



# Problem

- At the end of the subroutine how does the processor know which instruction to return to?
- This problem is solved by using a 'link register'.
- A link register holds the memory address of the instruction in the main program to which the subroutine returns to.
- Register r14 in the ARM microprocessor is designated as the link register and 'r14' can be replaced by 'lr' in mnemonics.

# Subroutines




- During the first pass through the subroutine the link register holds the return address 0x8008 and during the second pass it holds the return address 0x8014.

# Branch and link

- The mnemonic for ‘branch and link’ is BL;
  - a simple branch with mnemonic B does not update the link register.
- To return from the subroutine the value in the link register is moved into the program counter;
  - MOV pc, lr.
- What happens if a branch and link occurs in a subroutine?
  - The value held in the link register will be overwritten by a new return address so before one subroutine calls another the link register value must be stored elsewhere.

# Nested subroutines

- Complicated programs will have several subroutines and some subroutines will call other subroutines
  - this is standard practice and it is known as nesting.
- If subroutine A calls subroutine B the link register can not store the return address for both subroutines at the same time.
- In order to preserve the return address of all subroutines an area of computer memory called the stack is used.
- The stack is a 'last in first out' queue.

# The stack

- The stack is a last in first out queue
  - that means whatever data was added to the stack (‘pushed’) last is taken from the stack (‘popped’) first.
  - E.g. if the values pushed onto a stack were 0x00FF, 0xFF00, 0xAAAA in that order then they would be popped from the stack in the reverse order.
- In memory the stack is held as a list:

top of stack 0xAAAA

0xFF00

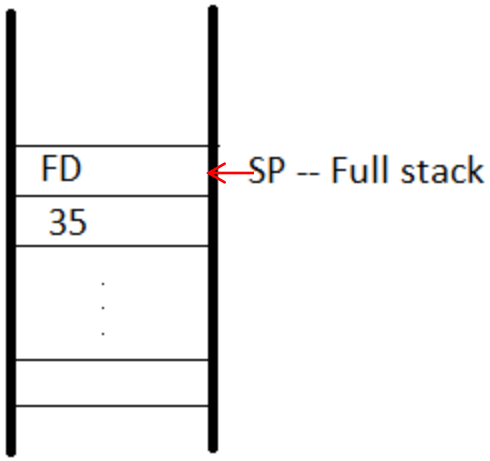
bottom of stack 0x00FF

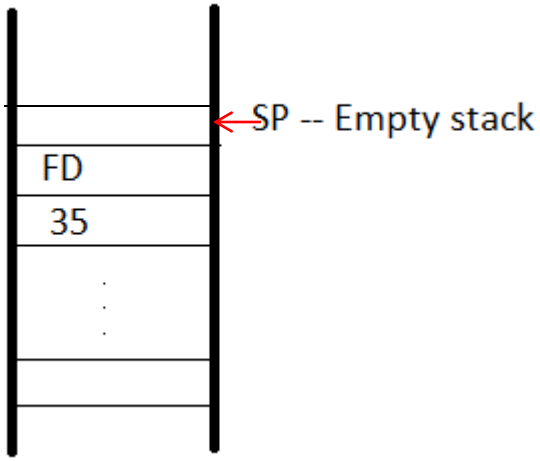
# Another problem

- The stack is a dynamic memory structure
  - Meaning that the amount of data held in the stack can vary.
- The bottom of the stack can be fixed at a particular memory address.
- How do we know where the top of the stack is?
  - We need to know so that we can pop the correct data.
- Another register is used to identify the top of the stack - it is known as the stack pointer.

# The stack pointer

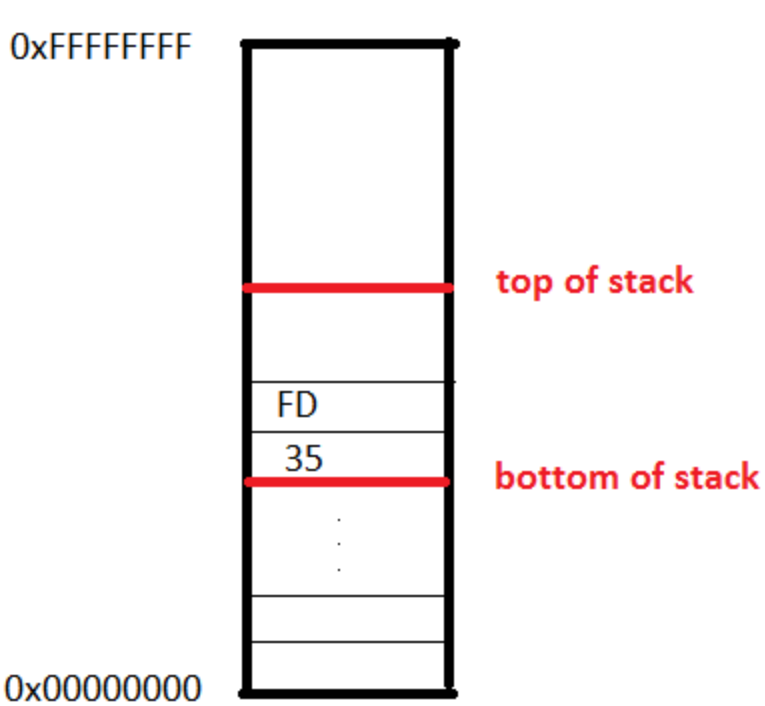
- The stack pointer holds an address in memory that identifies the top of the stack.
- The address is
  - either the location of the last data to be pushed onto the stack
  - or alternatively the location of the next empty slot where the next data can be placed.
- These two alternatives are known as a 'full stack' and an 'empty stack' and the ARM microprocessor allows both kinds.
  - Once a program uses one type of stack
  - switching to the other type is very unwise.


FD
35
SP -- Full stack

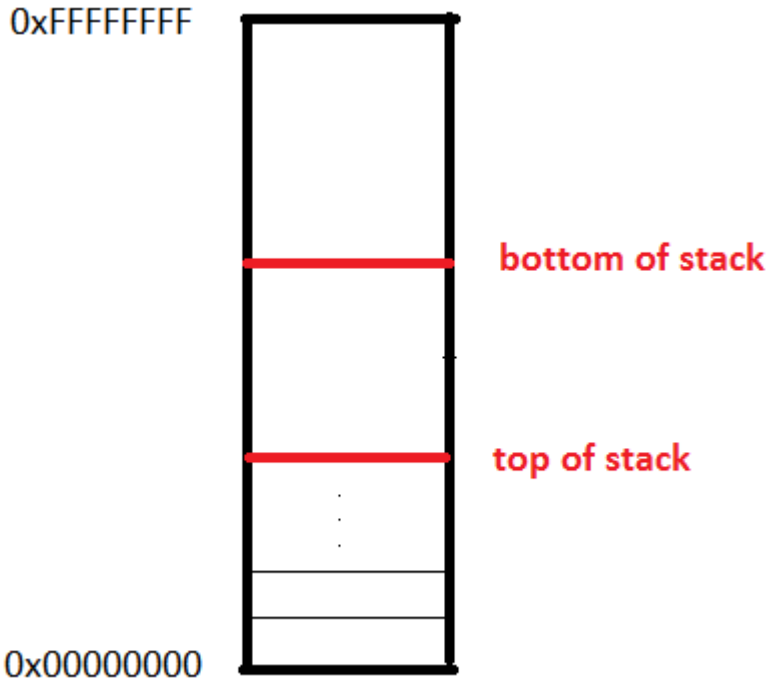

SP -- Empty stack
FD
35

# Ascending and descending stacks

- Stacks can also be either ascending or descending.
- For an ascending stack
  - the memory address of the top of the stack is greater than the memory address for the bottom of the stack.
  - When data is pushed onto an ascending stack the value held by the stack pointer increases and
  - When data is popped from an ascending stack it decreases.
- Descending stacks work in the opposite way
  - so that the top is at a lower memory address than the bottom.




Ascending stack




Descending stack

# Stacks and the ARM7

- Again the ARM allows both ascending and descending stacks
  - but once one type is used it is very unwise to switch to the other.
- Register r13 is used as the stack pointer and it can be replaced in mnemonics with 'sp'.
- To push the link register value onto a full descending stack the mnemonic is:
  - STMFD sp!, {lr}
- To pop a value from a full descending stack back into the link register the mnemonic is:
  - LDMFD sp!, {lr}

# Stacks and the ARM7

- So the first instruction in a subroutine is generally

  STMFD sp!, {lr}

- Then any branch and link in that subroutine can overwrite the link register.
- The subroutine would end by popping the return address from the stack into the link register

  LDMFD sp!, {lr}

  – followed by moving value in the link register into the program counter

  MOV pc, lr.

- It is common practice to replace these two instruction with one

  LDMFD sp!, {pc}

  – to pop the return address directly into the program counter.

# Stacking other registers

- Pushing and popping the stack can be achieved with several register at the same time
  - the mnemonics LDM and STM stand for load multiple and store multiple.
- Again if a subroutine overwrites any register, not just the link register, then the value held in that register can be pushed onto the stack at the start of the subroutine and can be popped from the stack at the end of the subroutine.
- E.g.
  - to push registers r6, r7, r8 and r9 with the link register use: STMFD sp!, {r6-r9, lr}
  - and to pop the same registers use: LDMFD sp!, {r6-r9, pc}

# Take care!

Great care must be taken when using push and pop to ensure that the number of pops is the same as the number of pushes.

E.g. for the following:

```
STMFD sp!, {r2, r4, r6, lr}
some instructions….
LDMFD sp!, {r2, r4, pc}
```

The program counter is loaded with the value previously held in register r6, not the return address previously held in the link register.

Order for Store: r15, r14, …, r0
Order for Load: r0, r1, …, r15

# The barrel shifter

- The barrel shifter is a very useful feature of the ARM7 microprocessor which allows bit patterns to be rotated.
- E.g. the instruction

  MOV r1, r2, LSL #5

  – would take the bit pattern in register r2 and shift it 5 places to the left before placing it in register r1. So if r2 held:

    1010 0101 1100 0011 1001 0110 1110 0111

  – after the instruction was executed r1 would hold:

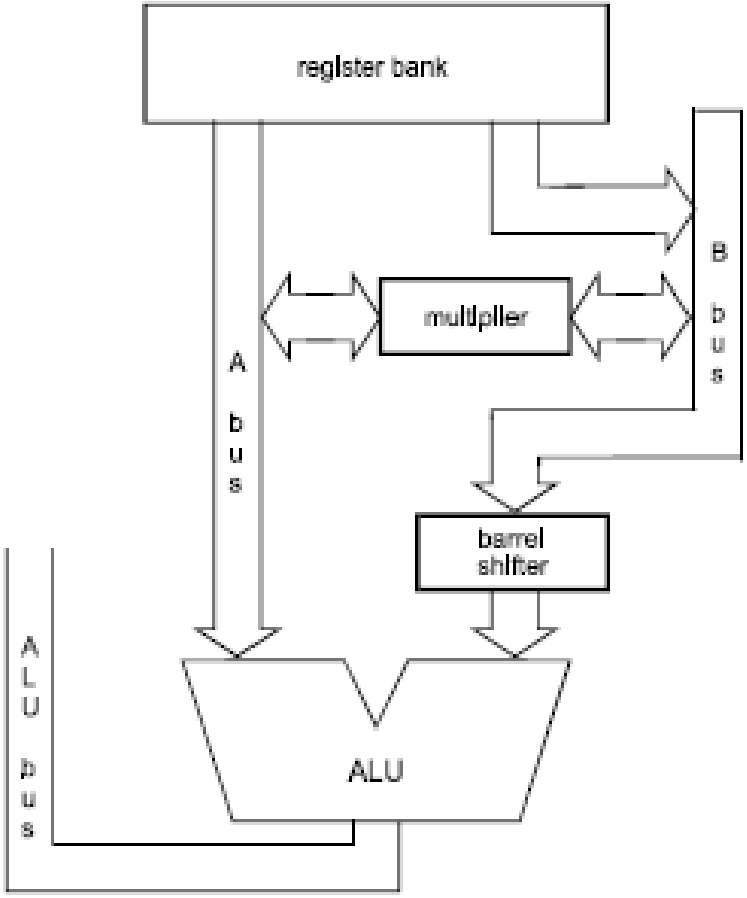
    1011 1000 0111 0010 1101 1100 1110 0000

# Arithmetic Logic Unit: ALU

The ALU is an important part of any microprocessor.

The ARM microprocessor divides the ALU functions into three blocks;

- a multiplier (that uses Booth's algorithm),
- the 'barrel shifter'
- and the rest of the ALU including the adder and logic functions. (This third block is generally referred to as the ALU.)



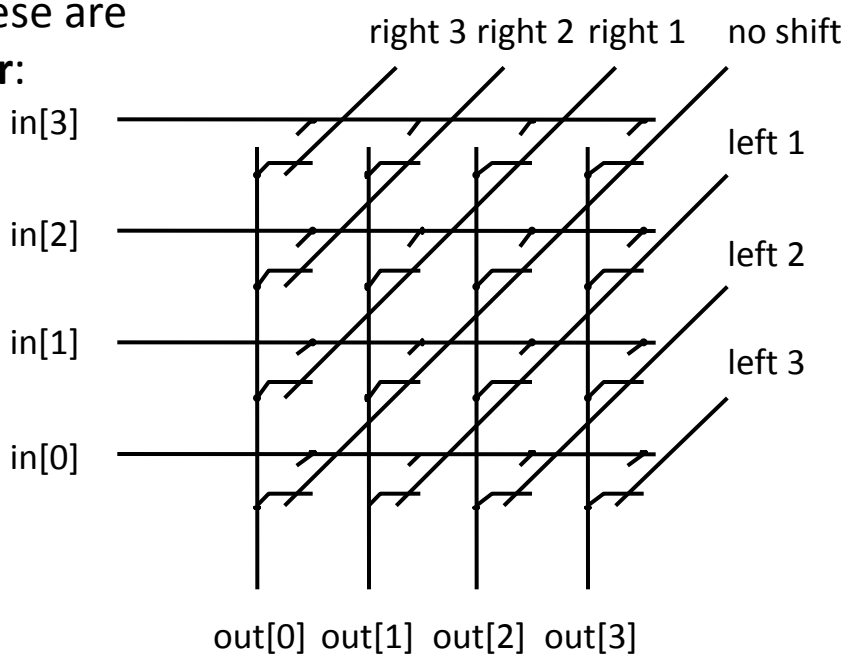
# Barrel Shifter

Shift and rotate are very important operations.

With integrated circuit techniques these are easily implemented by a **barrel shifter**:

There are different types of shift e.g.

- logical shift left (LSL),
- logical shift right (LSR)
- and arithmetic shift right (ASR).



# The ARM barrel shifter

The barrel shifter is a very useful feature of the ARM microprocessor which allows bit patterns to be rotated.

E.g. the instruction

MOV r1, r2, LSL #5

would take the bit pattern in register r2 and shift it 5 places to the left before placing it in register r1.

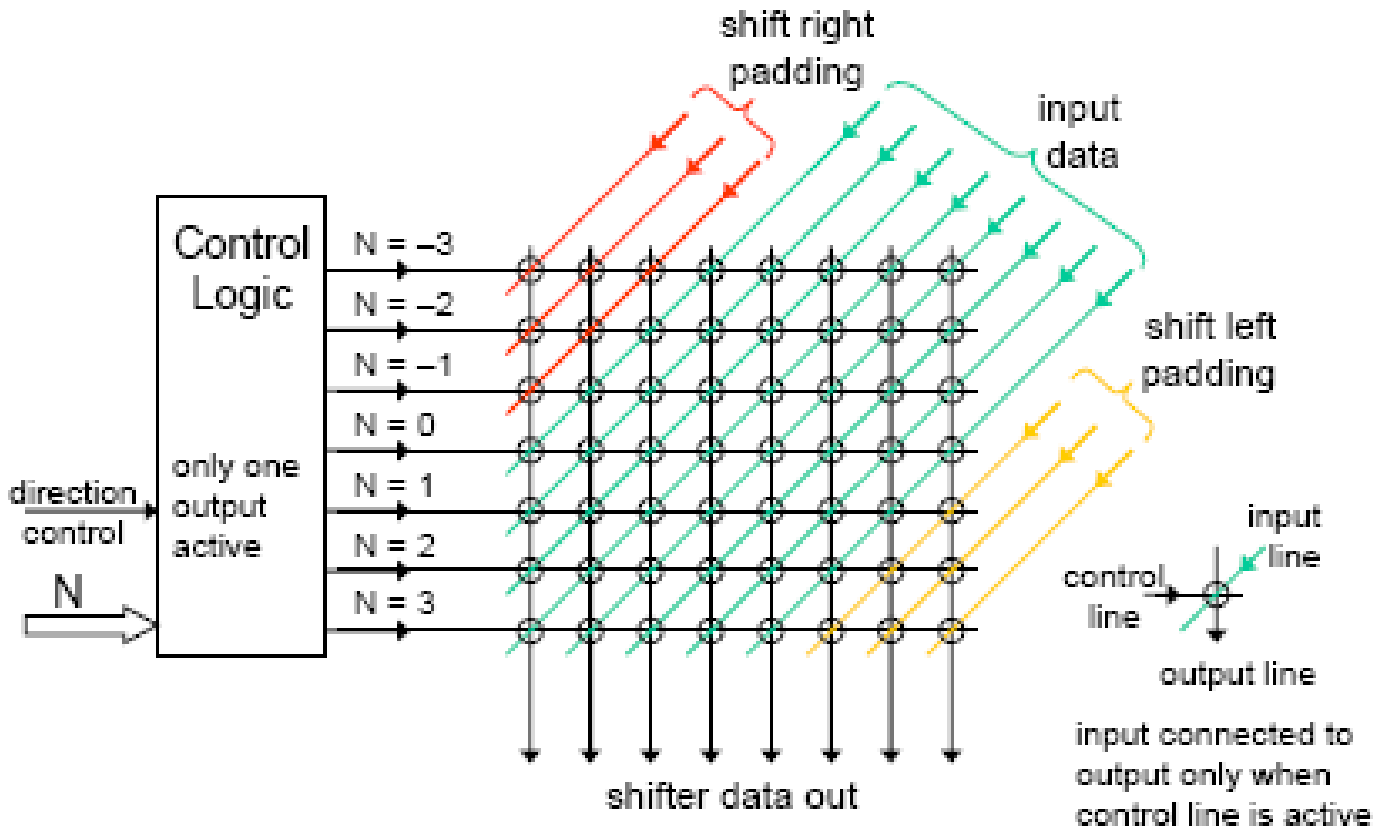
So if r2 held:

1010 0101 1100 0011 1001 0110 1110 0111

after the instruction was executed r1 would hold:

1011 1000 0111 0010 1101 1100 1110 0000

# Barrel Shifter (8 bits)

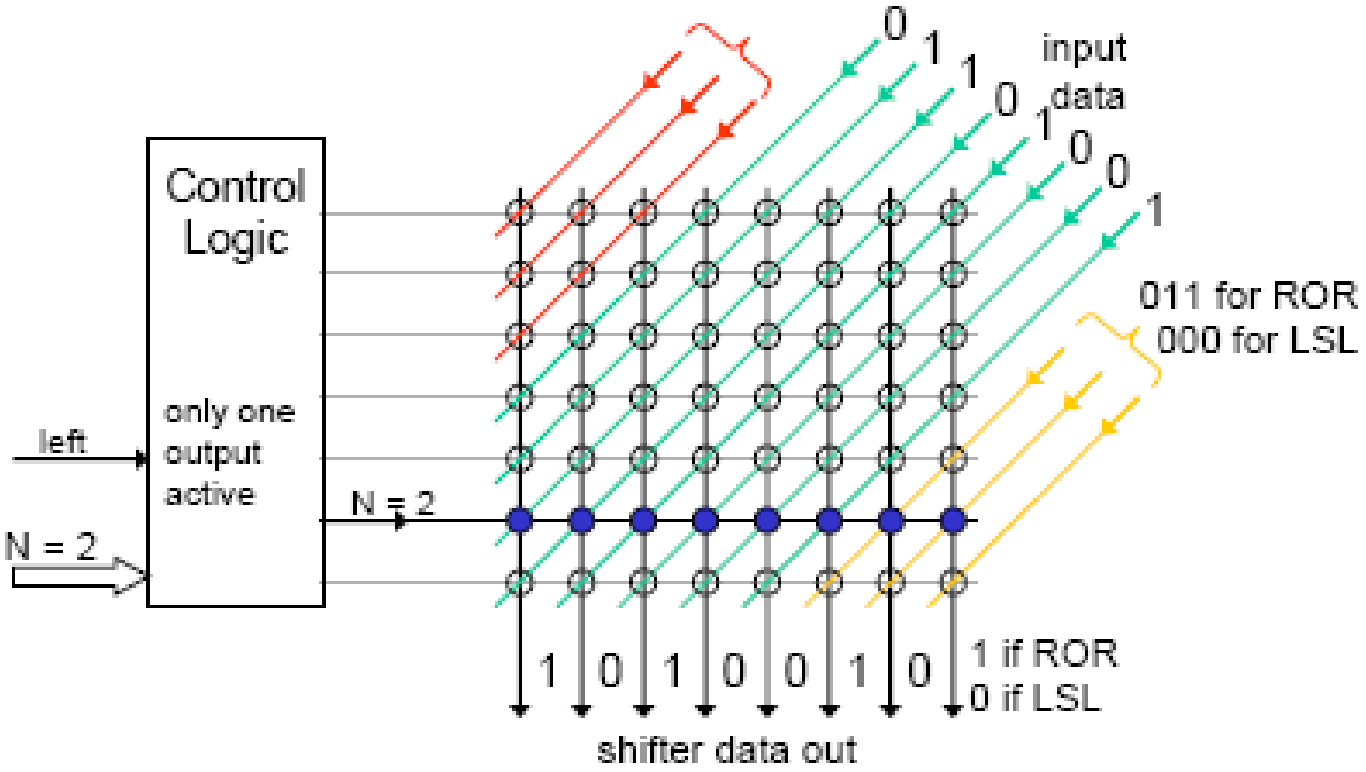


# Different shifts

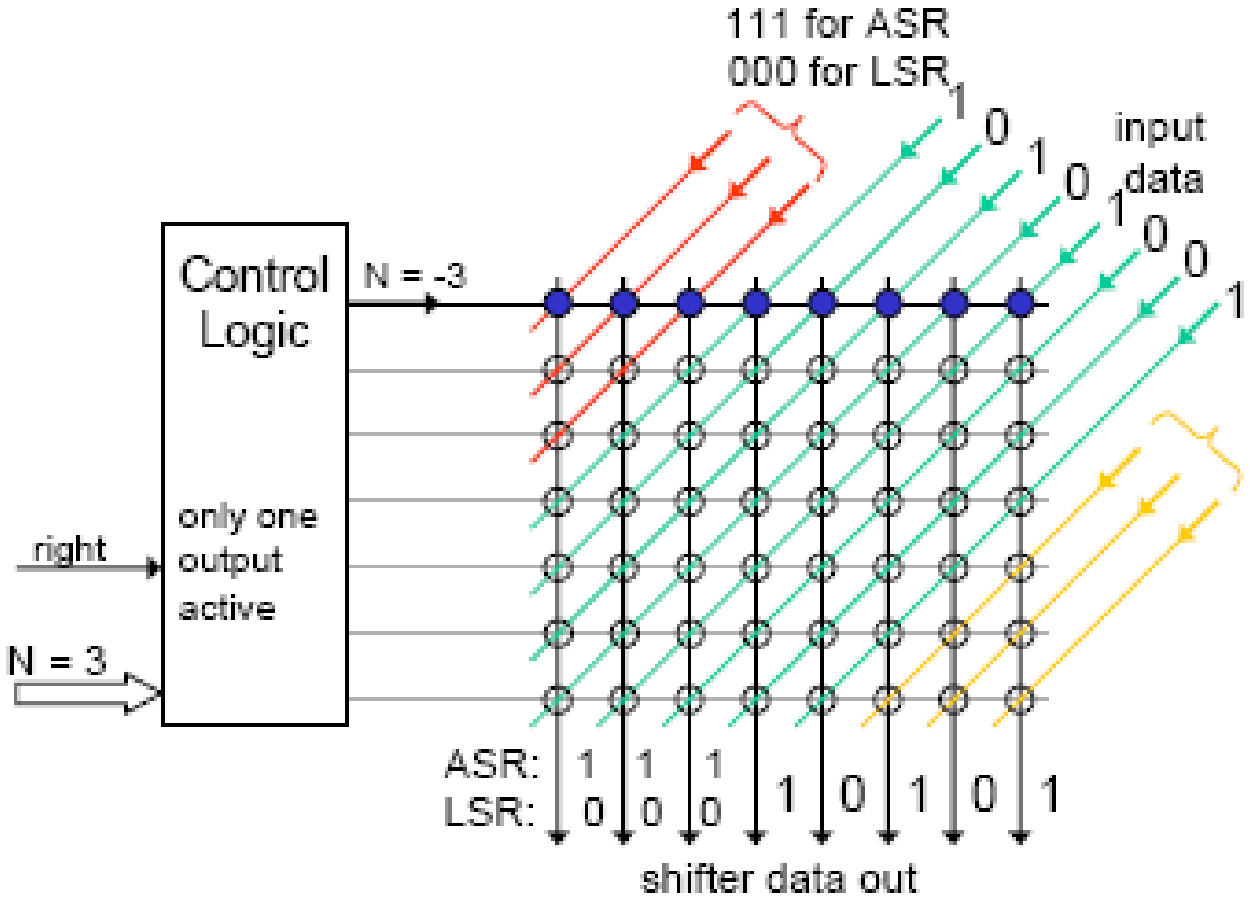
The ARM has five different types of shift:

- Logical shift left (LSL) - the bits are shifted to the left and the new bits added in at the right hand side are 0.
- Logical shift right (LSR) - the bits are shifted to the right and the new bits added in at the left hand side are 0.
- Arithmetic shift right (ASR) - bits are shifted rightwards and the new bits added in are the same as the ‘old’ sign bit; that is the msb of the input.
- Rotate right (ROR) - bits are rotated rightwards so that the bits shifted out at the right hand side reappear at the left hand side.
- Rotate extended (RRX) - bits are shifted right one place only and the carry flag is shifted into the new most significant bit. The least significant bit is shifted into the carry flag only if the mnemonic specifies an S.

# Example: Shift left by 2



# Example: Shift right by 3



# Shifts with other instructions

The ARM barrel shifter is placed in the datapath so that it can be used with many instructions such as:

MOV, ADD, ADC, SUB, RSB, AND, EOR, ORR, BIC

The shift is done first before the output of the barrel shifter is passed onto the ALU so that the instruction:

ADD r2, r3, r5, LSL #1

performs the following operation:

- the value in r5 is shifted left once (in effect doubling it's value)
- and then it is added to r3
- and the sum placed in r2.

# Using shift and add to multiply

Logical shift left can be used with MOV and ADD to multiply by $2^n$ and $2^n$ +1 E.g.

MOV rx, ry, LSL #n ; multiplies value in ry by $2^n$
ADD rx, ry, ry, LSL #n ; multiplies value in ry by $2^n$+1

E.g. multiply by 64 is given by MOV rx, ry, LSL #6

and multiply by 17 is given by ADD rx, ry, ry, LSL #4

SUB and RSB can also be used.

# Using shift to do integer division

Arithmetic shift right can be used to do an integer division by $2^n$ E.g.

MOV rx, ry, ASR #n ; divides value in ry by $2^n$

E.g. n=3, divide by 8
ry:= 0000……001111101000 (= $1{,}000_{10}$)
rx:= 0000000……001111101 (= $125_{10}$)

This also works in two's complement.
ry:= 1111……11110000011000 (= $-1{,}000_{10}$)
rx:= 1111111……11110000011 (= $-125_{10}$)