REGISTER TRANSFER AND MICROOPERATIONS

- Register Transfer Language
- Register Transfer
- Bus and Memory Transfers
- Arithmetic Microoperations
- Logic Microoperations
- Shift Microoperations
- Arithmetic Logic Shift Unit

SIMPLE DIGITAL SYSTEMS

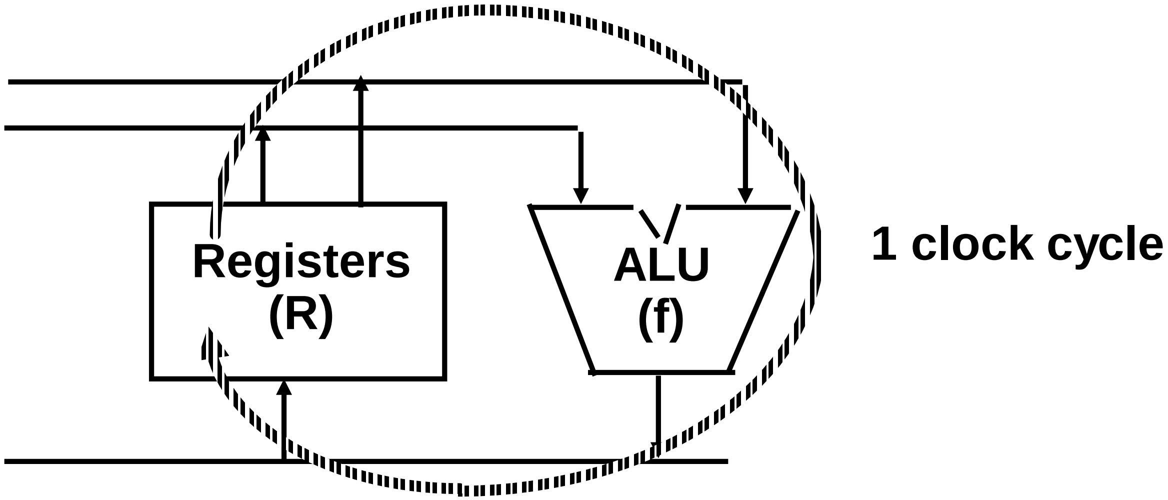
- **Combinational and sequential circuits (learned in Chapters 1 and 2) can be used to create simple digital systems.**
- **These are the low-level building blocks of a digital computer.**
- **Simple digital systems are frequently characterized in terms of**
  - **the registers they contain, and**
  - **the operations that they perform.**
- **Typically,**
  - **What operations are performed on the data in the registers**
  - **What information is passed between registers**

MICROOPERATIONS (1)

- **The operations on the data in registers are called microoperations.**
- **The functions built into registers are examples of microoperations**
  - **Shift**
  - **Load**
  - **Clear**
  - **Increment**
  - **…**

MICROOPERATION (2)

An elementary operation performed (during one clock pulse), on the information stored in one or more registers



$R \leftarrow f(R, R)$

f: shift, load, clear, increment, add, subtract, complement, and, or, xor, …

ORGANIZATION OF A DIGITAL SYSTEM

- **Definition of the (internal) organization of a computer**

  - Set of registers and their functions

  - Microoperations set

  Set of allowable microoperations provided by the organization of the computer

  - Control signals that initiate the sequence of microoperations (to perform the functions)

REGISTER TRANSFER LEVEL

- **Viewing a computer, or any digital system, in this way is called the register transfer level**

- **This is because we're focusing on**
  - **The system's registers**
  - **The data transformations in them, and**
  - **The data transfers between them.**

REGISTER TRANSFER LANGUAGE

- **Rather than specifying a digital system in words, a specific notation is used, *register transfer language***
- **For any function of the computer, the register transfer language can be used to describe the (sequence of) microoperations**
- **Register transfer language**
  - **A symbolic language**
  - **A convenient tool for describing the internal organization of digital computers**
  - **Can also be used to facilitate the design process of digital systems.**

DESIGNATION OF REGISTERS

- **Registers are designated by capital letters, sometimes followed by numbers (e.g., A, R13, IR)**
- **Often the names indicate function:**
  - **MAR - memory address register**
  - **PC - program counter**
  - **IR - instruction register**

- **Registers and their contents can be viewed and represented in *various ways***
  - **A register can be viewed as a single entity:**

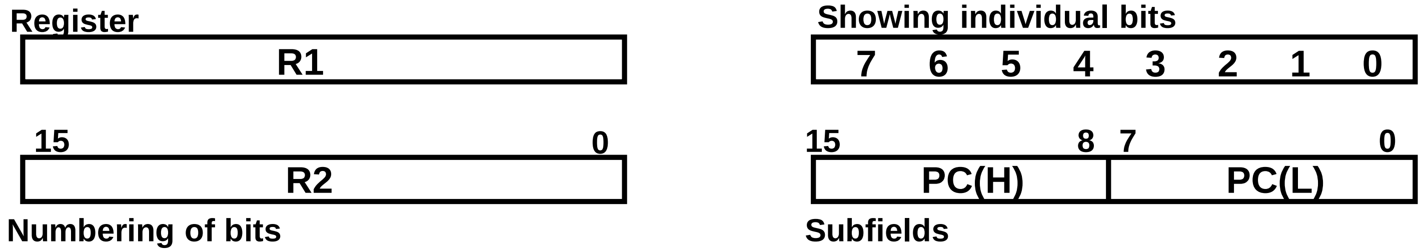
MAR

  - **Registers may also be represented showing the bits of data they contain**

DESIGNATION OF REGISTERS

- **Designation of a register**
  - **a register**
  - **portion of a register**
  - **a bit of a register**

- **Common ways of drawing the block diagram of a register**



REGISTER TRANSFER

- Copying the contents of one register to another is a register transfer
- A register transfer is indicated as

$R2 \leftarrow R1$

  - In this case the contents of register R1 are copied (loaded) into register R2
  - A simultaneous transfer of all bits from the source R1 to the destination register R2, during one clock pulse
  - Note that this is a non-destructive; i.e. the contents of R1 are not altered by copying (loading) them to R2

REGISTER TRANSFER

- **A register transfer such as**

  R3 $\leftarrow$ R5

  Implies that the digital system has

  - **the data lines from the source register (R5) to the destination register (R3)**
  - **Parallel load in the destination register (R3)**
  - **Control lines to perform the action**

CONTROL FUNCTIONS

- **Often actions need to only occur if a certain condition is true**
- **This is similar to an “if” statement in a programming language**
- **In digital systems, this is often done via a *control signal*, called a *control function***
  - **If the signal is 1, the action takes place**
- **This is represented as:**

P: R2 $\leftarrow$ R1

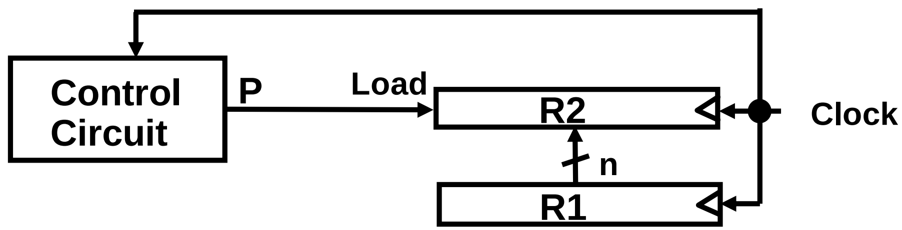
Which means “if P = 1, then load the contents of register R1 into register R2”, i.e., if (P = 1) then (R2 $\leftarrow$ R1)

HARDWARE IMPLEMENTATION OF CONTROLLED TRANSFERS

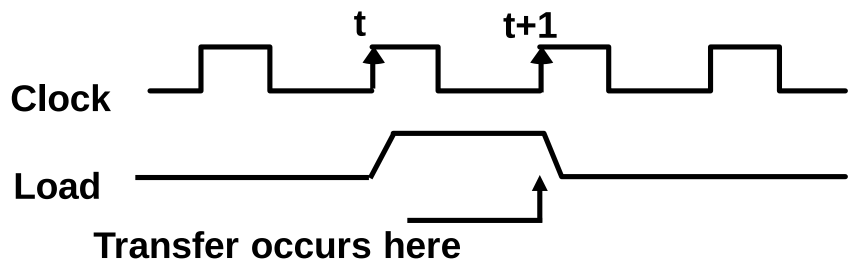
Implementation of controlled transfer

P: R2 $\leftarrow$ R1

Block diagram



Timing diagram



- **The same clock controls the circuits that generate the control function and the destination register**
- **Registers are assumed to use *positive-edge-triggered* flip-flops**

SIMULTANEOUS OPERATIONS

- **If two or more operations are to occur simultaneously, they are separated with commas**

P: R3 $\leftarrow$ R5, MAR $\leftarrow$ IR

- **Here, if the control function P = 1, load the contents of R5 into R3, and at the same time (clock), load the contents of register IR into register MAR**

BASIC SYMBOLS FOR REGISTER TRANSFERS

Symbols	Description	Examples
Capital letters & numerals	Denotes a register	MAR, R2
Parentheses ()	Denotes a part of a register	R2(0-7), R2(L)
Arrow $\leftarrow$	Denotes transfer of information	R2 $\leftarrow$ R1
Colon :	Denotes termination of control function	P:
Comma ,	Separates two micro-operations	A $\leftarrow$ B, B $\leftarrow$ A

CONNECTING REGISTRS

- **In a digital system with many registers, it is impractical to have data and control lines to directly allow each register to be loaded with the contents of every possible other registers**

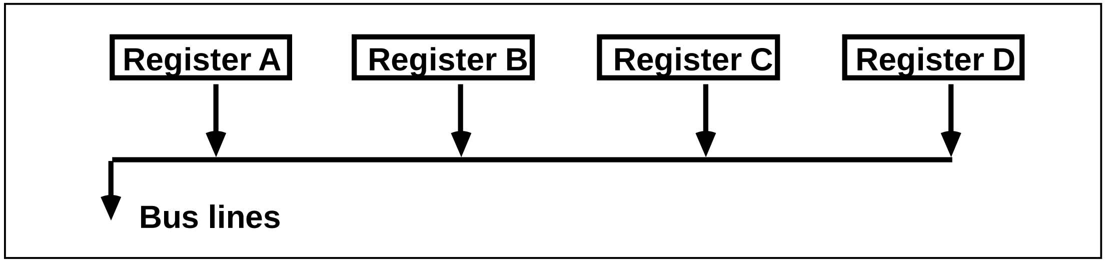
- **To completely connect n registers → n(n-1) lines**
- **O(n^2) cost**
  - **This is not a realistic approach to use in a large digital system**

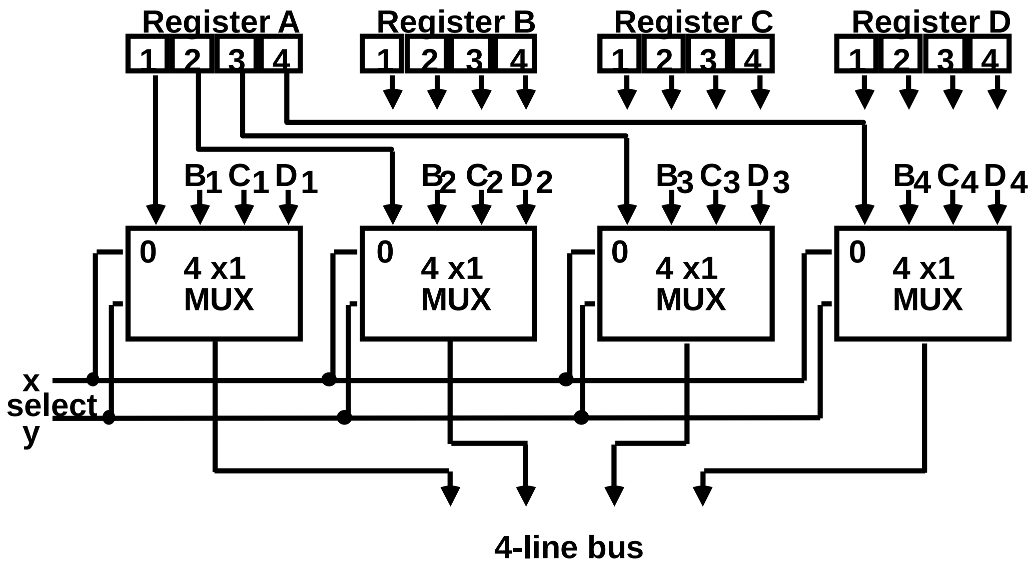
- **Instead, take a different approach**
- **Have one centralized set of circuits for data transfer – the bus**
- **Have control circuits to select which register is the source, and which is the destination**

BUS AND BUS TRANSFER

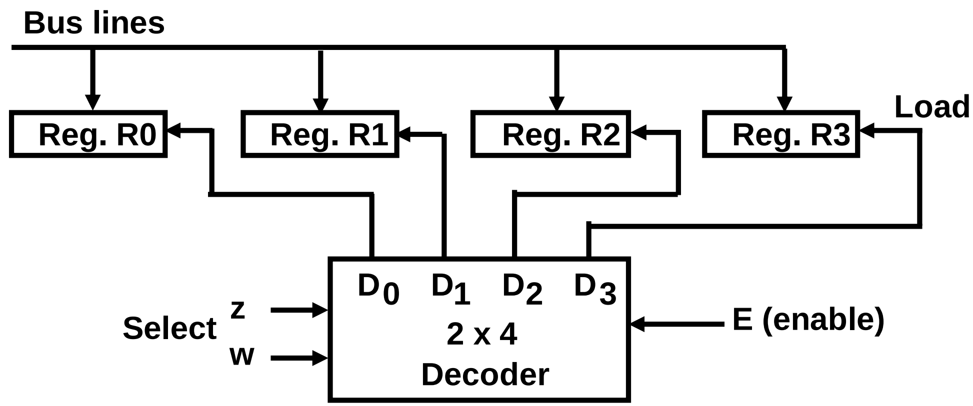
Bus is a path(of a group of wires) over which information is transferred, from any of several sources to any of several destinations.

From a register to bus: BUS $\leftarrow$ R

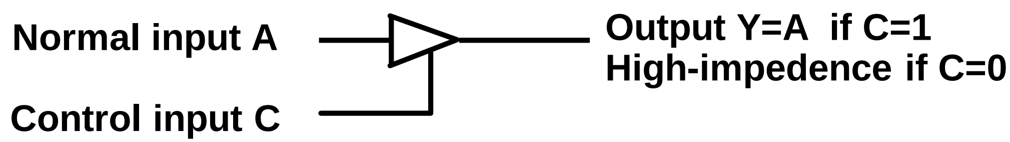




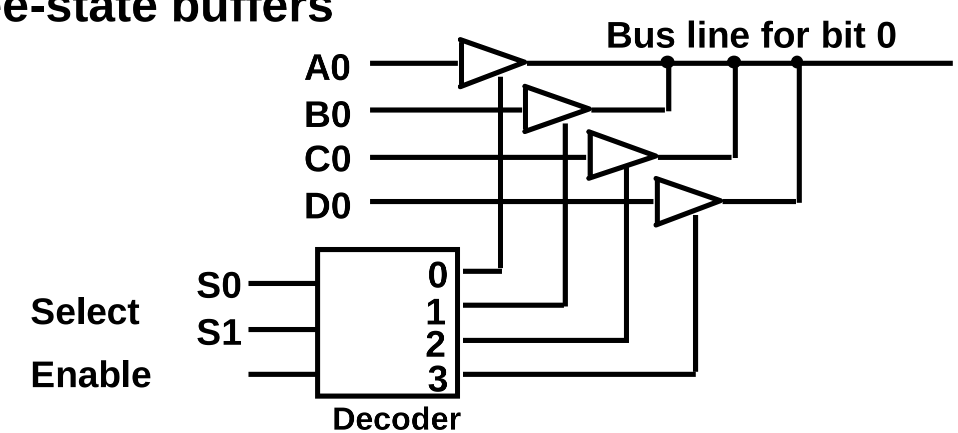
TRANSFER FROM BUS TO A DESTINATION REGISTER



Three-State (Bus) Buffers



Bus line with three-state buffers



BUS TRANSFER IN RTL

- **Depending on whether the bus is to be mentioned explicitly or not, register transfer can be indicated as either**

$$R2 \leftarrow R1$$

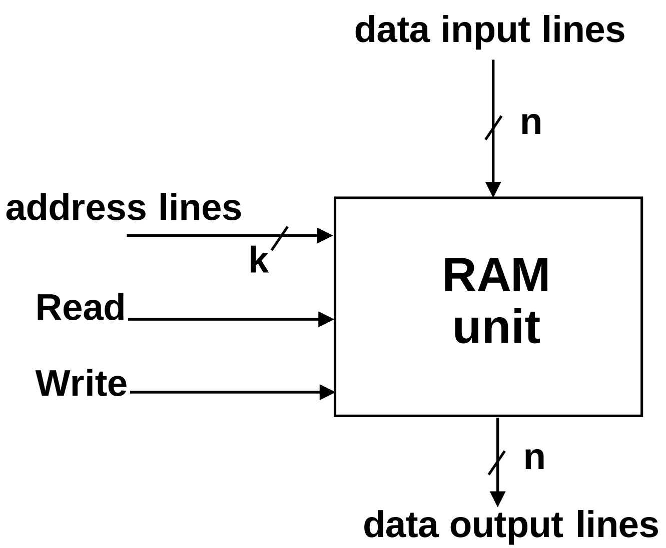
or

$$BUS \leftarrow R1, R2 \leftarrow BUS$$

- **In the former case the bus is implicit, but in the latter, it is explicitly indicated**

MEMORY (RAM)

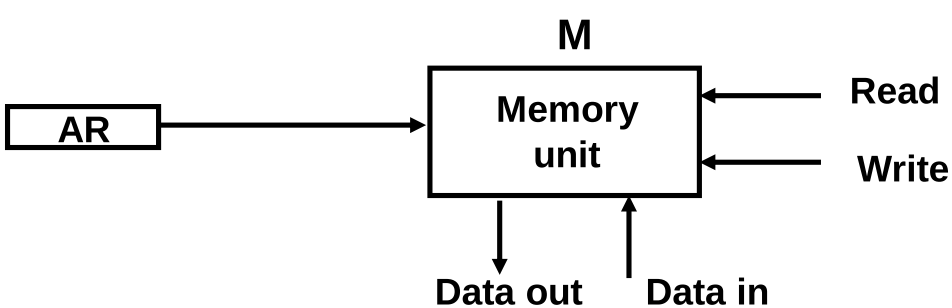
- **Memory (RAM) can be thought as a sequential circuit containing some number of registers**
- **These registers hold the *words* of memory**
- **Each of the r registers is indicated by an *address***
- **These addresses range from 0 to r-1**
- **Each register (word) can hold n bits of data**
- **Assume the RAM contains r = 2^k words. It needs the following**
  - n data input lines
  - n data output lines
  - k address lines
  - A Read control line
  - A Write control line



MEMORY TRANSFER

- **Collectively, the memory is viewed at the register level as a device, M.**
- **Since it contains multiple locations, we must specify which address in memory we will be using**
- **This is done by indexing memory references**

- **Memory is often accessed in computer systems by putting the desired address in a special register, the *Memory Address Register* (*MAR*, or *AR*)**
- **When memory is accessed, the contents of the MAR get sent to the memory unit's address lines**



MEMORY READ

- To read a value from a location in memory and load it into a register, the register transfer language notation looks like this:

$$R1 \leftarrow M[MAR]$$

- This causes the following to occur
  - The contents of the MAR get sent to the memory address lines
  - A Read (= 1) gets sent to the memory unit
  - The contents of the specified address are put on the memory's output data lines
  - These get sent over the bus to be loaded into register R1

MEMORY WRITE

- To write a value from a register to a location in memory looks like this in register transfer language:

$$M[MAR] \leftarrow R1$$

- This causes the following to occur
  - The contents of the MAR get sent to the memory address lines
  - A Write (= 1) gets sent to the memory unit
  - The values in register R1 get sent over the bus to the data input lines of the memory
  - The values get loaded into the specified address in the memory

SUMMARY OF R. TRANSFER MICROOPERATIONS

$A \leftarrow B$	Transfer content of reg. B into reg. A
$AR \leftarrow DR(AD)$	Transfer content of AD portion of reg. DR into reg. AR
$A \leftarrow$ constant	Transfer a binary constant into reg. A
$ABUS \leftarrow R1,$ $R2 \leftarrow ABUS$	Transfer content of R1 into bus A and, at the same time, transfer content of bus A into R2
AR	Address register
DR	Data register
M[R]	Memory word specified by reg. R
M	Equivalent to M[AR]
$DR \leftarrow M$	Memory *read* operation: transfers content of memory word specified by AR into DR
$M \leftarrow DR$	Memory *write* operation: transfers content of DR into memory word specified by AR

MICROOPERATIONS

- **Computer system microoperations are of four types:**

  - Register transfer microoperations

  - Arithmetic microoperations

  - Logic microoperations

  - Shift microoperations

ARITHMETIC MICROOPERATIONS

- **The basic arithmetic microoperations are**
  - Addition
  - Subtraction
  - Increment
  - Decrement

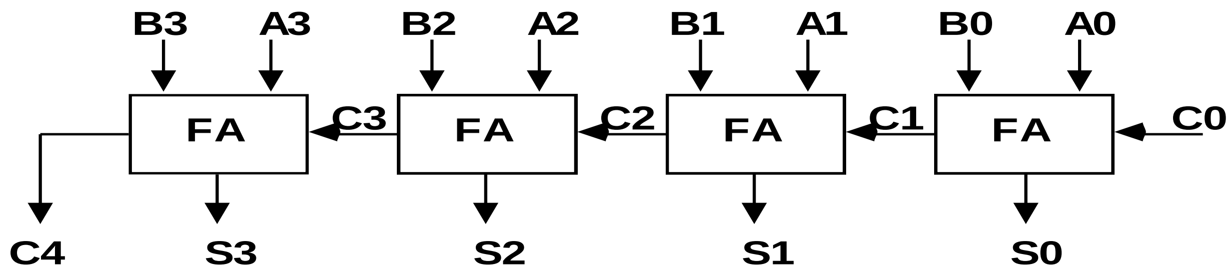
- **The additional arithmetic microoperations are**
  - Add with carry
  - Subtract with borrow
  - Transfer/Load
  - etc. …

Summary of Typical Arithmetic Micro-Operations

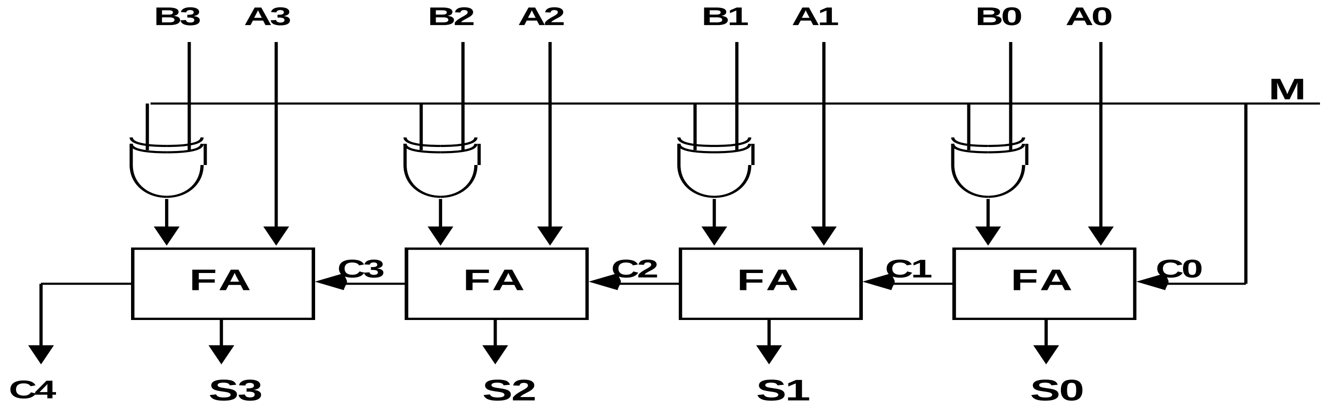
$R3 \leftarrow R1 + R2$	Contents of R1 plus R2 transferred to R3
$R3 \leftarrow R1 - R2$	Contents of R1 minus R2 transferred to R3
$R2 \leftarrow R2'$	Complement the contents of R2
$R2 \leftarrow R2' + 1$	2's complement the contents of R2 (negate)
$R3 \leftarrow R1 + R2' + 1$	subtraction
$R1 \leftarrow R1 + 1$	Increment
$R1 \leftarrow R1 - 1$	Decrement

BINARY ADDER / SUBTRACTOR / INCREMENTER

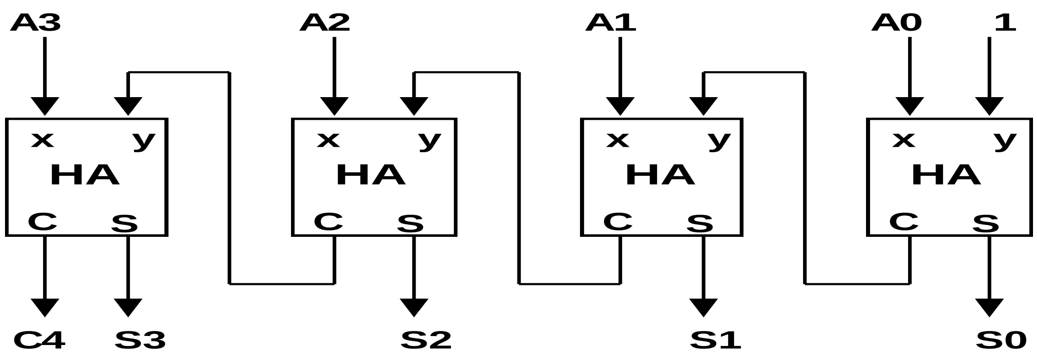
Binary Adder



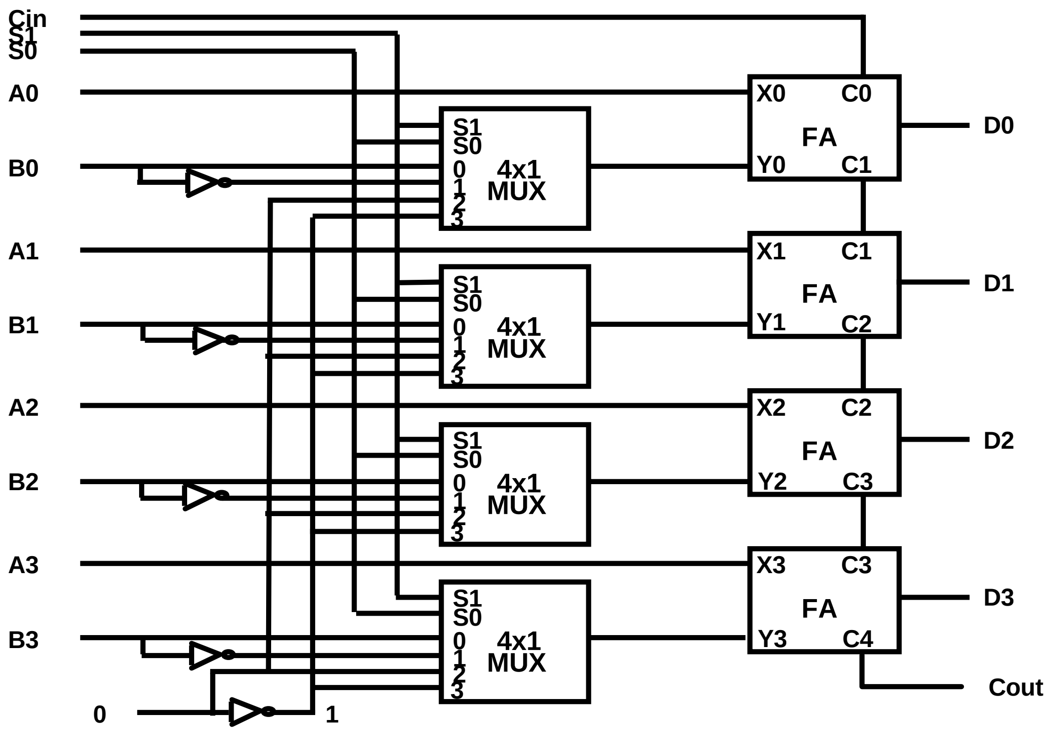
Binary Adder-Subtractor



Binary Incrementer



ARITHMETIC CIRCUIT



S1	S0	Cin	Y	Output	Microoperation
0	0	0	B	D = A + B	Add
0	0	1	B	D = A + B + 1	Add with carry
0	1	0	B’	D = A + B’	Subtract with borrow
0	1	1	B’	D = A + B’+ 1	Subtract
1	0	0	0	D = A	Transfer A
1	0	1	0	D = A + 1	Increment A
1	1	0	1	D = A - 1	Decrement A
1	1	1	1	D = A	Transfer A

LOGIC MICROOPERATIONS

- **Specify binary operations on the strings of bits in registers**
  - **Logic microoperations are bit-wise operations, i.e., they work on the individual bits of data**
  - **useful for bit manipulations on binary data**
  - **useful for making logical decisions based on the bit value**
- **There are, in principle, 16 different logic functions that can be defined over two binary input variables**

A	B	F_0	F_1	F_2	…	F_{13}	F_{14}	F_{15}
0	0	0	0	0	…	1	1	1
0	1	0	0	0	…	1	1	1
1	0	0	0	1	…	0	1	1
1	1	0	1	0	…	1	0	1

- **However, most systems only implement four of these**
  - **AND ($\wedge$), OR ($\vee$), XOR ($\oplus$), Complement/NOT**
- **The others can be created from combination of these**

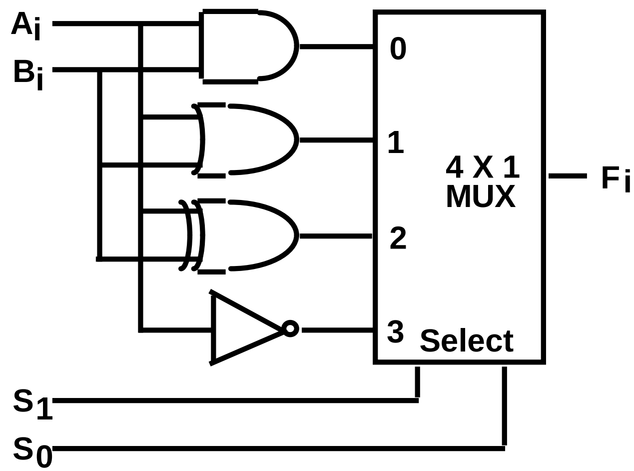
LIST OF LOGIC MICROOPERATIONS

- **List of Logic Microoperations**
  - 16 different logic operations with 2 binary vars.
  - n binary vars $\rightarrow 2^{2^n}$ functions

- **Truth tables for 16 functions of 2 variables and the corresponding 16 logic micro-operations**

x y	0 0 1 1 0 1 0 1	*Boolean Function*	*Micro-Operations*	*Name*
	0 0 0 0	F0 = 0	$F \leftarrow 0$	Clear
	0 0 0 1	F1 = xy	$F \leftarrow A \wedge B$	AND
	0 0 1 0	F2 = xy'	$F \leftarrow A \wedge B'$	
	0 0 1 1	F3 = x	$F \leftarrow A$	Transfer A
	0 1 0 0	F4 = x'y	$F \leftarrow A' \wedge B$	
	0 1 0 1	F5 = y	$F \leftarrow B$	Transfer B
	0 1 1 0	F6 = $x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
	0 1 1 1	F7 = x + y	$F \leftarrow A \vee B$	OR
	1 0 0 0	F8 = (x + y)'	$F \leftarrow (A \vee B)'$	NOR
	1 0 0 1	F9 = $(x \oplus y)'$	$F \leftarrow (A \oplus B)'$	Exclusive-NOR
	1 0 1 0	F10 = y'	$F \leftarrow B'$	Complement B
	1 0 1 1	F11 = x + y'	$F \leftarrow A \vee B'$	
	1 1 0 0	F12 = x'	$F \leftarrow A'$	Complement A
	1 1 0 1	F13 = x' + y	$F \leftarrow A' \vee B$	
	1 1 1 0	F14 = (xy)'	$F \leftarrow (A \wedge B)'$	NAND
	1 1 1 1	F15 = 1	$F \leftarrow$ all 1's	Set to all 1's

HARDWARE IMPLEMENTATION OF LOGIC MICROOPERATIONS



Function table

S_1	S_0	Output	μ-operation
0	0	$F = A \wedge B$	AND
0	1	$F = A \vee B$	OR
1	0	$F = A \oplus B$	XOR
1	1	$F = A'$	Complement

APPLICATIONS OF LOGIC MICROOPERATIONS

- **Logic microoperations can be used to manipulate individual bits or a portions of a word in a register**

- **Consider the data in a register A. In another register, B, is bit data that will be used to modify the contents of A**

  - **Selective-set** $A \leftarrow A + B$
  - **Selective-complement** $A \leftarrow A \oplus B$
  - **Selective-clear** $A \leftarrow A \bullet B'$
  - **Mask (Delete)** $A \leftarrow A \bullet B$
  - **Clear** $A \leftarrow A \oplus B$
  - **Insert** $A \leftarrow (A \bullet B) + C$
  - **Compare** $A \leftarrow A \oplus B$
  - **. . .**

SELECTIVE SET

- **In a selective set operation, the bit pattern in B is used to *set* certain bits in A**

1 1 0 0	A_t	
1 0 1 0	**B**	
1 1 1 0	A_{t+1}	$(A \leftarrow A + B)$

- **If a bit in B is set to 1, that same position in A gets set to 1, otherwise that bit in A keeps its previous value**

SELECTIVE COMPLEMENT

- **In a selective complement operation, the bit pattern in B is used to *complement* certain bits in A**

$$
\begin{array}{ll}
1\,1\,0\,0 & A_t \\
1\,0\,1\,0 & B \\
\hline
0\,1\,1\,0 & A_{t+1} \quad (A \leftarrow A \oplus B)
\end{array}
$$

- **If a bit in B is set to 1, that same position in A gets complemented from its original value, otherwise it is unchanged**

SELECTIVE CLEAR

- **In a selective clear operation, the bit pattern in B is used to *clear* certain bits in A**

1 1 0 0	A_t	
1 0 1 0	**B**	
0 1 0 0	A_{t+1}	$(A \leftarrow A \cdot B')$

- **If a bit in B is set to 1, that same position in A gets set to 0, otherwise it is unchanged**

MASK OPERATION

- **In a mask operation, the bit pattern in B is used to *clear* certain bits in A**

$$\begin{array}{ll} 1\,1\,0\,0 & A_t \\ 1\,0\,1\,0 & B \\ \hline 1\,0\,0\,0 & A_{t+1} \quad (A \leftarrow A \cdot B) \end{array}$$

- **If a bit in B is set to 0, that same position in A gets set to 0, otherwise it is unchanged**

CLEAR OPERATION

- **In a clear operation, if the bits in the same position in A and B are the same, they are cleared in A, otherwise they are set in A**

1 1 0 0	A_t	
1 0 1 0	B	
0 1 1 0	A_{t+1}	$(A \leftarrow A \oplus B)$

INSERT OPERATION

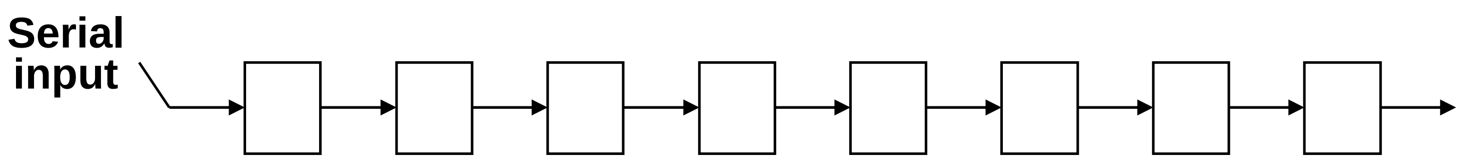
- **An insert operation is used to introduce a specific bit pattern into A register, leaving the other bit positions unchanged**
- **This is done as**
  - **A mask operation to clear the desired bit positions, followed by**
  - **An OR operation to introduce the new bits into the desired positions**
  - **Example**
    - » **Suppose you wanted to introduce 1010 into the low order four bits of A: 1101 1000 1011 0001 A (Original)**
      1101 1000 1011 1010 A (Desired)

```
» 1101 1000 1011 0001        A (Original)
  1111 1111 1111 0000        Mask
  ---------------------------------------------
  1101 1000 1011 0000        A (Intermediate)
  0000 0000 0000 1010        Added bits
  ---------------------------------------------
  1101 1000 1011 1010        A (Desired)
```

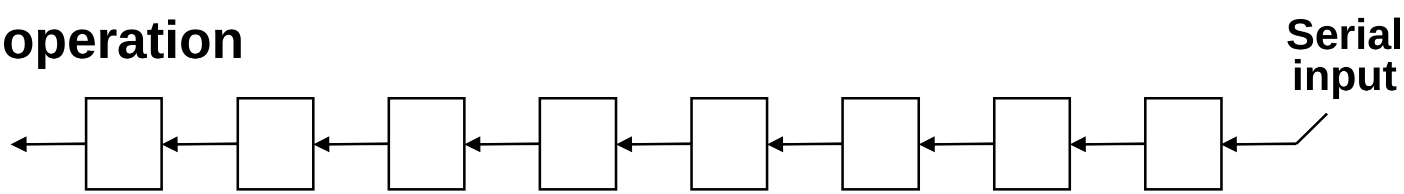
SHIFT MICROOPERATIONS

- **There are three types of shifts**
  - ***Logical shift***
  - ***Circular shift***
  - ***Arithmetic shift***
- **What differentiates them is the information that goes into the serial input**

- **A right shift operation**



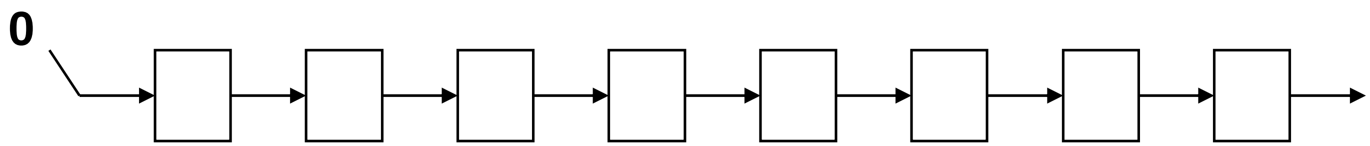
- **A left shift operation**



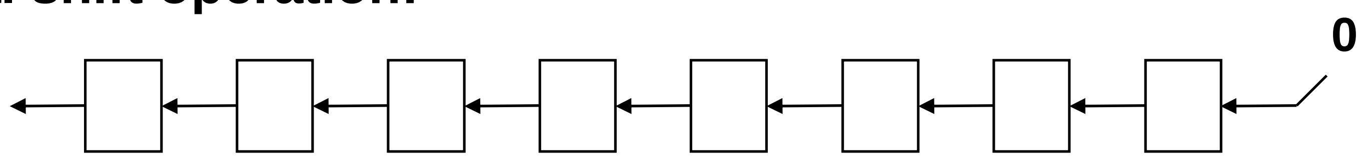
LOGICAL SHIFT

- In a logical shift the serial input to the shift is a 0.

- A right logical shift operation:



- A left logical shift operation:

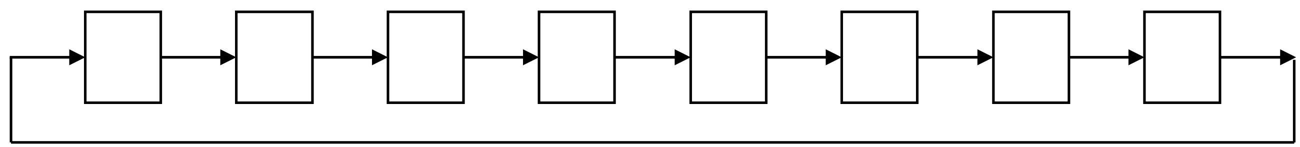


- In a Register Transfer Language, the following notation is used
  - *shl* for a logical shift left
  - *shr* for a logical shift right
  - Examples:
    - » R2 ← *shr* R2
    - » R3 ← *shl* R3

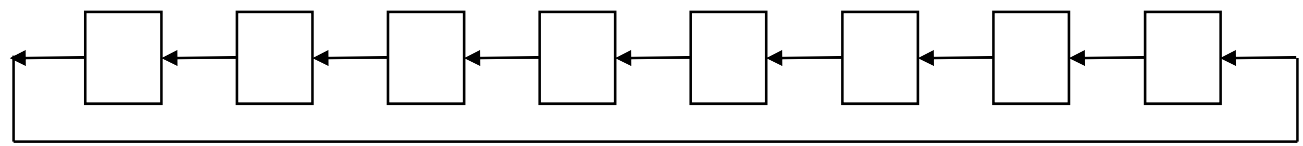
CIRCULAR SHIFT

- In a circular shift the serial input is the bit that is shifted out of the other end of the register.

- A right circular shift operation:



- A left circular shift operation:

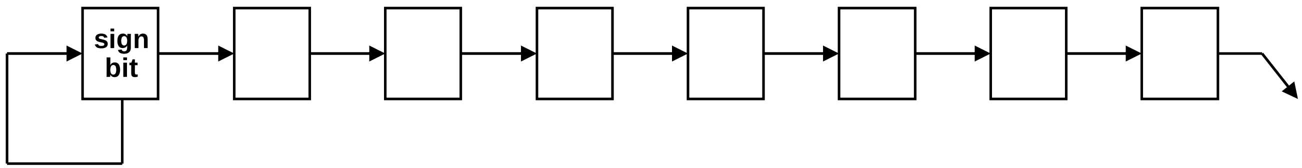


- In a RTL, the following notation is used
  - *cil* for a circular shift left
  - *cir* for a circular shift right
  - Examples:
    - » R2 ← *cir* R2
    - » R3 ← *cil* R3

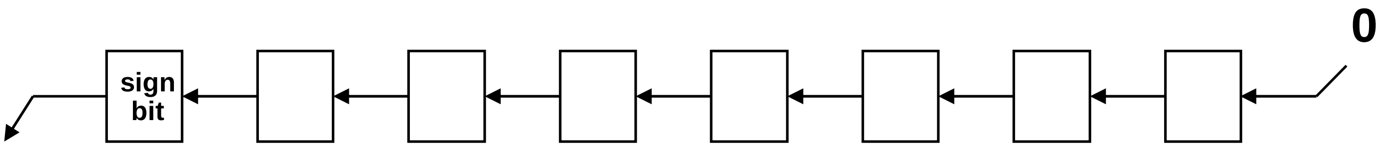
ARITHMETIC SHIFT

- **An arithmetic shift is meant for signed binary numbers (integer)**
- **An arithmetic left shift multiplies a signed number by two**
- **An arithmetic right shift divides a signed number by two**
- **The main distinction of an arithmetic shift is that it must keep the sign of the number the same as it performs the multiplication or division**

- **A right arithmetic shift operation:**

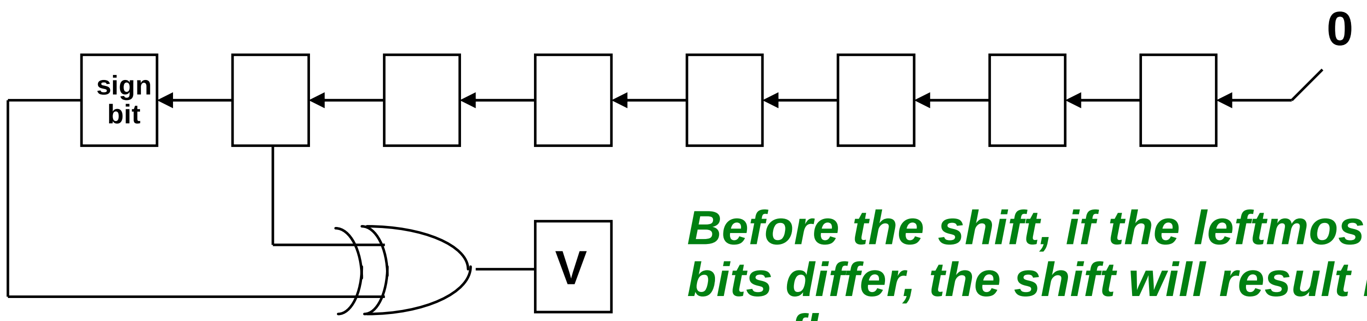


- **A left arithmetic shift operation:**



ARITHMETIC SHIFT

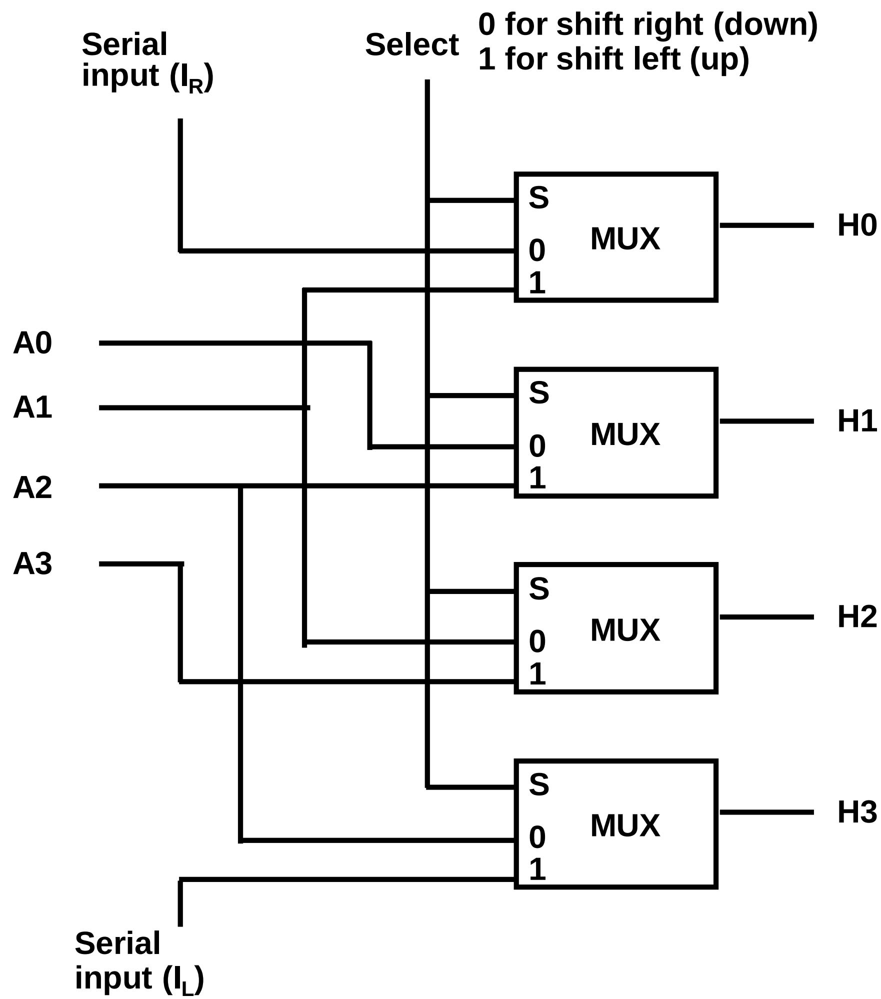
- **A left arithmetic shift operation must be checked for the overflow**



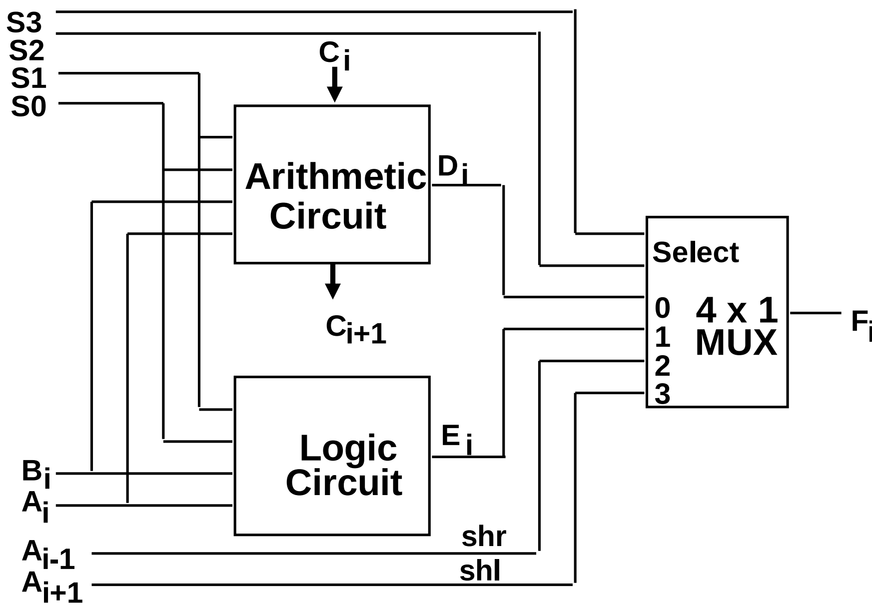
Before the shift, if the leftmost two bits differ, the shift will result in an overflow

- **In a RTL, the following notation is used**
  - ***ashl*** **for an arithmetic shift left**
  - ***ashr*** **for an arithmetic shift right**
  - **Examples:**
    - » **R2 $\leftarrow$ *ashr* R2**
    - » **R3 $\leftarrow$ *ashl* R3**

HARDWARE IMPLEMENTATION OF SHIFT MICROOPERATIONS



ARITHMETIC LOGIC SHIFT UNIT



S3	S2	S1	S0	Cin	Operation	Function
0	0	0	0	0	F = A	Transfer A
0	0	0	0	1	F = A + 1	Increment A
0	0	0	1	0	F = A + B	Addition
0	0	0	1	1	F = A + B + 1	Add with carry
0	0	1	0	0	F = A + B'	Subtract with borrow
0	0	1	0	1	F = A + B'+ 1	Subtraction
0	0	1	1	0	F = A - 1	Decrement A
0	0	1	1	1	F = A	TransferA
0	1	0	0	X	$F = A \wedge B$	AND
0	1	0	1	X	$F = A \vee B$	OR
0	1	1	0	X	$F = A \oplus B$	XOR
0	1	1	1	X	F = A'	Complement A
1	0	X	X	X	F = shr A	Shift right A into F
1	1	X	X	X	F = shl A	Shift left A into F