SCS1207	COMPUTER ARCHITECTURE AND ORGANISATION	L	T	P	Credits	Total Marks
		3	0	0	3	100

COURSE OBJECTIVES

- To understand the organization of a computer, and the hardware - software interface.
- To know about the various components of a computer and their internals.
- To have an overview of pippelinig, vector processing and multiprocessors.

UNIT 1 INTRODUCTION **10 Hrs.**

Central Processing unit - Introduction - General Register Organization - Stack organization - Basic computer Organization - Instruction codes - Computer Registers - Computer Instructions - Instruction Cycle - Arithmetic – Logic - Shift Microoperations - Arithmetic Logic Shift unit - Example Architectures: MIPS – Power – PC – RISC - CISC

UNIT 2 DATA PATH DESIGN **10 Hrs.**

Computer arithmetic : Addition - Subtraction - Multiplication and Division algorithms - Floating Point Arithmetic operations

Microprogrammed Control : Control memory - address sequencing - Microprogram Example - Design of Control unit - Example Processor design

UNIT 3 MEMORY ORGANISATION **8 Hrs.**

Memory Organization : Memory Hierarchy - Main memory - auxiliary Memory - Associative Memory - Cache Memory - Virtual memory

UNIT 4 IO ORGANISATION **9 Hrs**

Input - Output Organization : Peripheral Devices - I/O Interface - Modes of transfer - Priority Interrupt - DMA - IOP - Serial Communication

UNIT 5 MULTIPROCESSORS **8 Hrs.**

Characteristics of multiprocessors - Interconnection Structures - Interprocessor Arbitration - Interprocessor Communication and Synchronization - Cache coherance

Max. 45 Hours

TEXT / REFERENCE BOOKS

1. M.Morris Mano, 'Computer system Architecture",Prentice-Hall Publishers,Third Edition.
2 John P Hayes , 'Computer architecture and Organization', McGraw Hill international edition, Third Edition.
3 Kai Hwang and Faye A Briggs ,'Computer Architecture and Parallel Processing', McGraw Hill international edition,1995.

END SEMESTER EXAM QUESTION PAPER PATTERN

Max. Marks : 80 **Exam Duration : 3 Hrs.**

PART A : 10 questions of 2 marks each- No choice **20 Marks**

PART B : 2 questions from each unit of internal choice, each carrying 12 marks **60 Marks**

UNIT I - INTRODUCTION

Central Processing unit (CPU):

Alternately referred to as a processor, central processor, or microprocessor, the CPU is the Central Processing Unit of the computer. A computer's CPU handles all instructions it receives from hardware and software running on the computer. In

terms of computing power, the CPU is the most important element of a computer system. It add and compare its data in CPU chip. A CPU or Processors of all computers, whether micro, mini or mainframe must have three element or parts primary storage, arithmetic logic unit (ALU), and control unit. Control Unit (CU) - decodes the program instruction. CPU chip used in a computer is partially made out of Silica, on other words silicon chip used for data processing are called Micro Processor. It is the brain that runs the show inside the PC. All work that is done on a computer is performed directly or indirectly by the processor. Obviously, it is one of the most important components of the Pc. It is also, scientifically, not only one of the most amazing parts of the PC, but one of the most amazing devices in the world of technology.

Major components of CPU:

In the CPU, the primary components are the ALU (Arithmetic Logic Unit) that performs mathematical, logical, and decision operations and the CU (Control Unit) that directs all of the processors operations.

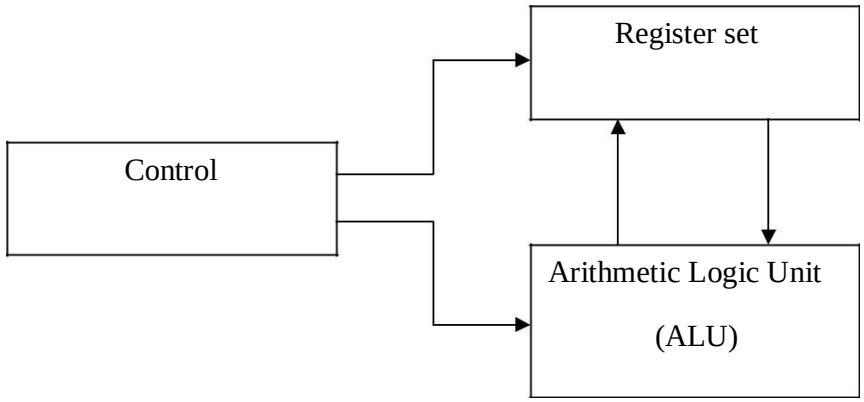


Fig. 1.1: Central Processing Unit (CPU)

In general, most processors are organized in one of 3 ways

Single register (Accumulator) organization

- Basic Computer is a good example

- Accumulator is the only general purpose register

Example - ADDX /* AC AC + M[X] */

General register organization

- Used by most modern computer processors
- Any of the registers can be used as the source or destination for computer operations

Example -

```
ADD R1, R2, R3      /* R1  R2 + R3 */
ADD R1, R2          /* R1  R1 + R2 */
MOVR1, R2           /* R1  R2 */
ADD R1, X           /* R1  R1 + M[X] */
```

Stack organization

- All operations are done using the hardware stack
- For example, an OR instruction will pop the two top elements from the stack, do a logical OR on them, and push the result on the stack

Example - PUSHX /* TOS M[X] */

We are interested with address field of instructions with multiple address fields in instructions. The number of address fields in the instruction format depends on the internal organization of CPU. Some CPU combines features from more of one structure.

1.2 General Register Organizations

Intermediate data are needed to be stored like pointers, counters, return address, temp results, and partial products.

Cannot save them in main memory because their access is time consuming.

It is more efficient and faster to be stored inside processor.

So the solution is designing multiple registers inside processor and connects them through a common bus.

In Basic Computer, there is only one general purpose register, the Accumulator (AC) but in modern CPUs, there are many general purpose registers.

It is advantageous to have many registers

Transfer between registers within the processor are relatively fast

Going “off the processor” to access memory is much slower

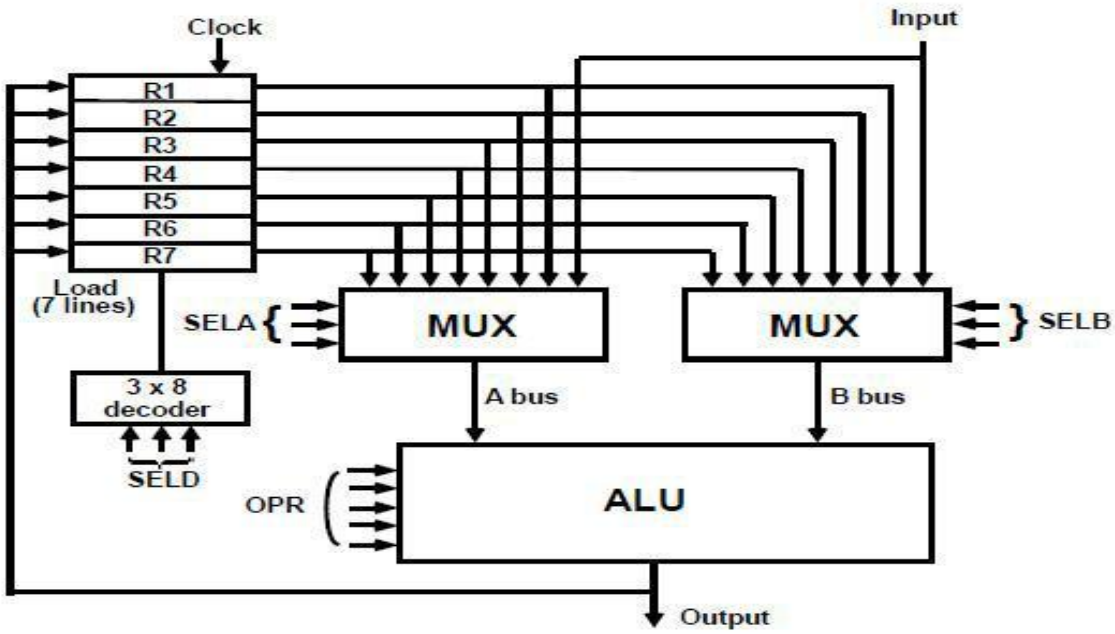
1.2.1 Register set with common ALU

A new bus organization will be introduced here in order to clarify the idea of register banking and how to control their actions.

We have 7 CPU registers that their outputs are connected to 2 MUX 8 X 1 to form the 2 buses A and B.

The A and B are inputted to ALU unit in which its operation is selected by their select lines among different arithmetic and logic operations.

The resulted ALU data can is directed to the input of all 7 registers which one of them will be selected according to 3 X 8 decoder connected to LD inputs of the register.



For example to perform operation

$$R1 = R2 + R3$$

The control then provides

MUXA select R2
MUXB select R3
OPR in ALU operation for ADD
SELD to direct destination register R1

These four control signals are generated in control unit in start of each clock cycle ensuring operands are selected beside correct ALU operation and result is chosen in one clock cycle only.

1.2.2 CONTROL WORD

There are 14 selection inputs in the unit and their combined value specifies control word.

3 bits to select A source, 3 bits to select B source, 5 bits to select operation required on them and finally 3 bits to select destination register. Encoding of 3 bits for selection of the 2 sources plus the destination is defined in next table. While the other table specifies ALU operations encoding.

Encoding of register selection bits

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

Encoding of ALU operations

OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	ADD A + B	ADD
00101	Subtract A - B	SUB
00110	Decrement A	DECA
01000	AND A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA
11000	Shift left A	SHLA

1.2.3 Arithmetic Logic Unit (ALU):

ALU provides arithmetic operations (ADD, SUB, INCA, DECA)
Logic operations (AND, OR, XOR, COMA)
Shift operations (SHLA SHRA).
And Transfer operation (TSFA)

Stack Organization:

Stack is a storage device that stores information in a way that the item is stored last is the first to be retrieved (LIFO).

Stack in computers is actually a memory unit with address register (stack pointer SP) that can count only. SP value always points at top item in stack.

The two operations done on stack are,

PUSH (Push Down), operation of insertion of items into stack

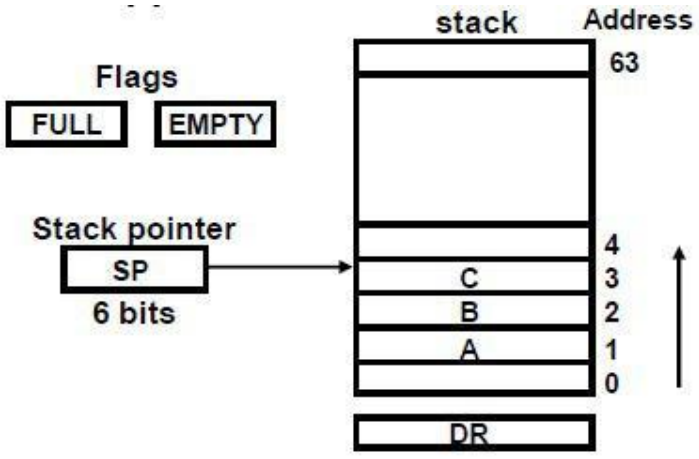
POP (Pop Up), operation of deletion item from stack

Those operation are simulated by INC and DEC stack register (SP).

1. Register stack

   A stand alone unit that consists of collection of finite number of registers.

   The next example shows 64 location stack unit with SP that stores address of the word that is currently on the top of stack.



Note that 3 items are placed in the stack A, B, and C. Item C is in top of stack so that SP holds 3 which the address of item C.

To remove top item from stack (popping stack) we start by reading content of address 3 and decrementing the content of SP. Item B is now in top of stack holding address 2.

To insert new item (pushing the stack) we start by incrementing SP then writing a new word where SP now points to (top of stack).

Note that in 64 word stack we need to have SP of 6 bits only (from 000000 to 111111). If 111111 is reached then at next push SP will be 000000, that is when the stack is FULL. Similarly when SP is 000001 then at next pop SP will go to 000000 that is when the stack is EMTY.

Initially, SP = 0, EMPTY = 1, FULL =0

Procedures for pushing stack

```
SP       SP + 1
M[SP]      DR
IF (SP = 0) THEN (FULL = 1)
EMTY  0
```

Note that:

1. Always we use DR to pass word into stack
2. M[SP] memory word specified by address currently in SP
3. First item stored in stack is at address 1
4. Last item stored in stack is at address 0. That is FULL = 1
5. Any push to stack means EMTY = 0

Procedures for popping stack

```
DR   M[SP]
SP  SP – 1
IF (SP = 0) THEN (EMTY = 1)
FULL  0
```

Note That:

1. Top of stack is read into DR
2. If SP reached 0 then stack is EMTY = 1. That when SP was 1 then pop occurred. No more pops can happen from here.
3. Any pop from stacks means FULL = 0

2. Memory Stack

Stack can be implemented in RAM memory attached to CPU. Only by assigning special part of it for stack operations.

Next figure shows of main memory divided into program, data, and stack.

PC points to next instruction in instruction part

AR points to array of data of operands

SP points to top of stack

All are connected to common address bus

Stack grows (pushed) with decreasing address and empties (pops) with increasing address.

New item is inserted with push operation by decrementing SP then a write to SP address is done

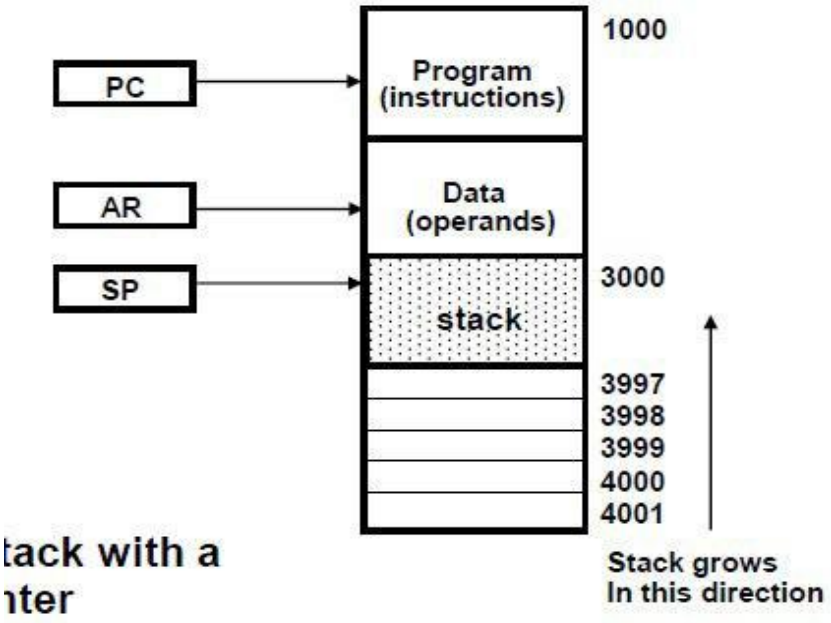
SP ← SP -1

M [SP] ← DR

Last item is removed from stack with pop operation by removing item by reading from memory location addressed by SP then SP is incremented.

DR ← M [SP]

SP ← SP +1



As shown in figure initial value of SP is 4001 and first item when pushed in stack stores at address 4000 and second one stores at address 3999. The last address pushed into will be 3000. (See limitation danger?)

Most computers are not supported by hardware to sense stack overflow and underflow. But can be implemented by saving the 2 limits in 2 registers. After each push or pop the SP is compared with the limit to see if stack has reached its limits. So must be taking care of using software.

Reverse Polish notations

Always in this way we load SP with bottom address of stack portion of memory
Very useful notation to utilize stacks to evaluate arithmetic expressions.

We write in infix notation such as:

A*B + C*D

We compute A*B, store product, compute C*D, then sum two products. So we have to scam back and forth to see which operation comes first.

The 3 notations to evaluate expressions

1. A + B Infix notation
2. +AB Prefix notation (Polish notation)
3. AB+ Postfix notation (reverse Polish

Reverse Polish Notation is in a form suitable for stack manipulation. Starts by scanning expression from left to right. When operator is found then perform

Instruction Format:

operation with 2 operands in left of operator and replace result place of 2 operands and operator. Then you can continue this until you reach final answer.

Example

Expression A*B + C*D is written in RPN as AB*CD*+. And will computed as

(A*B) CD *+

(A*B)(C*D)+

Example

Convert infix notation expression (A + B)*(C * (D + E) + F) to RPN?

AB+ DE+ C * F+ *.

Will be computed as

(A+B) (D+E) C * F + *

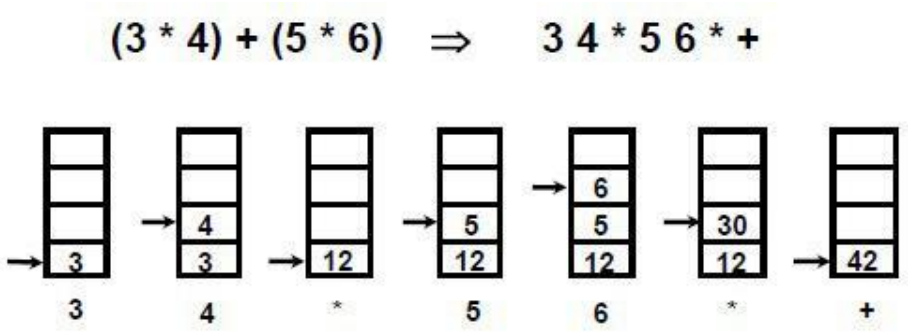
Reverse polish notation combined with stack comprised of registers is most efficient way to evaluate expression. Stacks are good for handling long and complex

problems involving chain calculations. But need first to convert arithmetic expressions into parenthesis-free reverse polish notation.

This procedure is employed in some scientific calculators and some computers. Example

Convert (3*4) (5*6) to RPN

34*56*+



Instruction format

The Instruction coding fields in today's computers follow the next format

1. Operation code field to specify operation
2. Address field that specifies operand address field or register
3. Mode field to specify effective address

1. Operation code field

Types of instruction based on operation

1. Data Transfer and Manipulation Instructions
2. Arithmetic Instructions
3. Logical Instructions
4. Shift instructions
5. Branch Instructions

1. Data Transfer and Manipulation:

Computers provide an extensive set of instructions to give the user the flexibility to execute different tasks.

Instructions sets of different processors differ from each other in mainly in the way operands are determined from address and mode fields. Even the op-code will be different among them as well.

Move data from one place to another. The most common transfer are between memory and processor registers, between processor registers and IO, and between processor registers.

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

- Load instruction is used to transfer data from memory to processor register(s) (Accumulator).
- Store instruction transfers data from register(s)(Accumulator) to memory.
- Move instruction is used to move data from registers and from register to memory and vice versa.
- Exchange instruction swaps data between 2 registers or between 2 memory locations.
- Input-Output instructions transfer data between processors and IO device
- Push-Pop instructions transfer data between stack and registers

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	AC ← M[ADR]
Indirect address	LD @ADR	AC ← M[M[ADR]]
Relative address	LD $ADR	AC ← M[PC + ADR]
Immediate operand	LD #NBR	AC ← NBR
Index addressing	LD ADR(X)	AC ← M[ADR + XR]
Register	LD R1	AC ← R1
Register indirect	LD (R1)	AC ← M[R1]
Autoincrement	LD (R1)+	AC ← M[R1], R1 ← R1 + 1
Autodecrement	LD -(R1)	R1 ← R1 - 1, AC ← M[R1]

Let's take the load instruction to show how different addressing modes can be used with this instruction. This table shows recommended assembly language convention and actual transfer done in each case.

Note that

ADR	address
NBR	number
X	index register
R1	processor register
AC	accumulator
@	indirect addressing
$	relative address to PC
#	immediate operand
( )	register indirect mode
+	auto increment combined with register indirect
-	auto decrement combined with register indirect

2. Arithmetic Instructions:

They will be the 4 basic operations: Add, Subtract, Multiply, and Divide.
Multiplication and Division usually generated using software subroutines.
Next table shows typical arithmetic instructions in general processors.

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with Carry	ADDC
Subtract with Borrow	SUBB
Negate(2's Complement)	NEG

ADD, SUB, MUL, DIV instructions may operate with different data types whether available in registers or memory. Like in integer type, floating Point type, and BCD type needs special instructions

ADDI add integers
ADDF add floating point
ADDD add in BCD

Since number of bits in registers is finite and hence resulted data are finite precision, some processors support hardware double precision operations arithmetic that occupies 2 words.

3. Logical and Bit manipulation instructions:

Logical instructions perform binary operations on bits stored in registers and maybe in memory.
Helpful for manipulating single bits or group of bits

Performs on single bits as separated from each other and treated as Boolean variable.

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

- Clear instruction forces all bits of operand to be 0
- Complement instruction inverts all bits of operand (0 to1, 1to 0). Carry can be set, clear, or complemented with special instructions.
- Interrupt Enable and Disable instructions are shown.

4. Shift instructions:

Shift operands instructions are useful.
Shifts are : logical shifts, arithmetic shifts, and rotate type instructions
Next table lists 4 types of shift instructions

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right thru carry	RORC
Rotate left thru carry	ROLC

Logical shifts insert 0 at the ends
Arithmetic shifts preserve the sign of operand in most cases but not in all cases (2's complement rule).
Arithmetic shift right preserves sign bit
Arithmetic shift left is the same as logical shift left.
Rotate instructions is a circular shift bit shifted out in one end is inserted in next end.
Rotates instructions may involves carry bit or not.

5. Branch Instructions (Program Control altered)

The program control instructions may change address in PC and cause the normal sequential execution to be. So this types of instruction causes breaks in execution sequence.

The next table lists some program control instructions.

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RTN
Compare(by –)	CMP
Test(by AND)	TST

Branch and jump instructions are usually the same and they execute immediate change of program sequence to another address.

Branch and Jump instructions may be conditional or unconditional. in conditional case specifies a condition (aero, positive, negative, greater than, and so on) if condition is met the execution transfers to new address. If not met then execution continues sequentially.

Skip instruction skips the instruction immediately after it and executes the next then next one. Conditional skip will do the skip if a condition is met

SKIP ON
COND BRA
AD1 BRA
AD2

Call and Return instructions are used with subroutines to jump to and come back from subroutines.

Compare instruction subtracts the 2 operand and change some flags in status register. Used to make conditional jumps afterword.

Test instruction performs AND between 2 operands and conditional flags will be changed accordingly.

Status Bit Conditions

ALU of any processor is equipped with condition code bits or flags. Next figure shows 8-bit ALU with 4-bit status flags (C, S, Z, and V). those can be set and cleared.

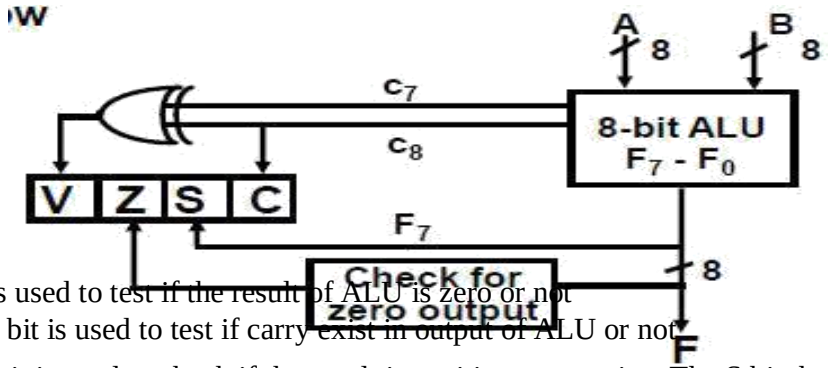
In Basic Computer, the processor had several (status) flags –1 bit value that indicated various information about the processor's state –E, FGI, FGO, I, IEN, R.

C (Carry): Set to 1 if the carry out of the ALU is 1
S (Sign): The MSB bit of the ALU‟s output
Z (Zero): Set to 1 if the ALU‟s output is all 0‟s
V (Overflow): Set to 1 if there is an overflow if last 2 carries = 1.
If output of ALU > 127 or < -128.

Conditional Branch Instructions:

The below table lists the most common branch instructions

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	Z = 1
BNZ	Branch if not zero	Z = 0
BC	Branch if carry	C = 1
BNC	Branch if no carry	C = 0
BP	Branch if plus	S = 0
BM	Branch if minus	S = 1
BV	Branch if overflow	V = 1
BNV	Branch if no overflow	V = 0
	Unsigned compare conditions (A - B)	
BHI	Branch if higher	A > B
BHE	Branch if higher or equal	A ≥ B
BLO	Branch if lower	A < B
BLOE	Branch if lower or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B
	Signed compare conditions (A - B)	
BGT	Branch if greater than	A > B
BGE	Branch if greater or equal	A ≥ B
BLT	Branch if less than	A < B
BLE	Branch if less or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B



Zero bit is used to test if the result of ALU is zero or not

The carry bit is used to test if carry exist in output of ALU or not

The sign bit is used to check if the result is positive or negative. The S bit the most significant bit of result

The overflow bit is used with arithmetic operations done on signed numbers.

For comparison and branches a subtraction of 2 operands must occur in advance A – B then the comparison branch follows.

CMP A, B CMP A, B CMP A, B CMP A, 0 CMP A, 0
BE ,BLT,BGT,BP,BN

When S=0	means	A >B	for signed
When S=1	means	A<B	for signed
When C=0	means	A>B	for unsigned
When C=1	means	A<B	for unsigned
When Z=1	means	A=B	

Subroutine Call and Return:

During execution a subroutine maybe called many times to perform given task at various point of the main program.

A subroutine call transfers control to a subroutine procedure. We can call it
Call subroutine

Jump to subroutine o Branch subroutine

Branch and save return address

When finished subroutine a return instruction returns address back to main program.

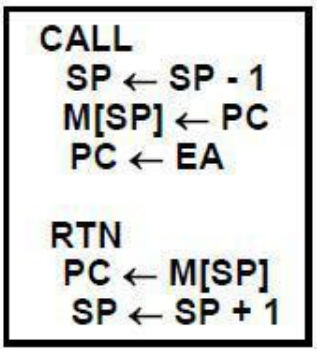
Now what will happen in a call subroutine

Branch to the beginning of the Subroutine-Same as the Branch or Conditional Branch.

Save the Return Address to get the address of the location in the Calling Program upon exit from the Subroutine. Some save return address in:

1. First location, of subroutine.
2. Fixed location in memory.
3. In fixed register.
4. In stack.

Next figure shows micro operations implementing calling/returning from subroutines



Mode Field : Addressing Modes:

The addressing mode specifies the rule for translating or modifying the address field of the instruction before the operand is fetched.

The way the operands are chosen during program execution is dependent on addressing modes.

Why computer used addressing modes more of them?

To give programming versatility by providing a way to implement counters, pointers, indexing of data, and program reallocation.

To reduce number of addressing fields of instruction

In some processors the addressing mode of the instruction is specified with distinct binary code; while in other processors, uses single binary code that designates the operation and its addressing mode.

1. Implied mode:

The operands are specified implicitly in the definition of the instruction.

No need to specify address in the instruction

All register reference instruction in Basic Computer that uses accumulator is from this type. Since registers holding operand(s) are implied in op code of the operation itself.

Zero address instructions in stack-organized computers are implied mode instruction since operands are implied always at top of stack.

Examples: CLA, CME, INP

2. Immediate mode:

Instead of specifying the address of the operand, operand itself is specified with the instruction.

No need to specify address in the instruction
However, operand itself needs to be specified
Sometimes, require more bits than the address
Fast to acquire an operand
Useful mode to initialize registers to constant values (initial).

3. Register mode:

Address specified in the instruction is the register address that resides within CPU.

Designated operand need to be in a register
Shorter address than the memory address
Saving address field in the instruction
Faster to acquire an operand than the memory
Addressing.

EA = IR(R) (IR(R): Register field of IR)

4. Register Indirect mode:

Instruction specifies a register which contains the memory address of the operand

Saving instruction bits since register address is shorter than the memory address and register is specifying the address here.

Slower to acquire an operand than both the register addressing or memory addressing

User must ensure that address of operand is already sited in mentioned register.

EA = [IR(R)] ([x]: Content of x)

5. Autoincrement/Autodecrement mode:

Similarto register indirect mode except that the Autoincrement/Autodecrement mode register is incremented or decremented
after or before its value is used to access memory.

Useful to point to next or previous data referenced to current data pointed by register. Therefore used in table access.

Automatically implement Increment/Decrement content of specified register.

6. Direct Address mode:

Instruction specifies the memory address which can be used directly to access the memory.

Faster than the other memory addressing modes since operand address comes with op code of the instruction.

(BAD)Too many bits are needed to specify the address for a large physical memory space.

In branch type instructions the address field specifies branch address.

EA = IR(addr) (IR(addr): address field of IR)

7. Indirect Address mode:

The address field of an instruction specifies the address of a memory location that contains the address of the operand.

When the abbreviated address is used large physical memory can be addressed with a relatively small number of bits.

Slow to acquire an operand because of an additional memory access

EA = M[IR(address)]

8. Relative Address mode:

The Address fields of an instruction specifies the part of the address (abbreviated address) which can be used along with a designated register to calculate the address of the operand

Address field of the instruction is short (few bits)

Large physical memory can be accessed with a small number of address bits EA = f(IR(address), R), R is sometimes implied

3 different Relative Addressing Modes depending on R

PC Relative Addressing Mode(R = PC)

EA = PC + IR(address)

Example: if PC=825 and address part in instruction =24. Then address branched to 826 + 24 = 850.

Indexed Addressing Mode(R = IX, where IX: Index Register)

EA = IX + IR(address)

Base Register Addressing Mode(R = BAR, where BAR: Base Address Register)

EA = BAR + IR(address)

9. Indexed Addressing mode:

The content of index register is added to address part of instruction to obtain Effective Address (EA).

We can see as address field of instruction specifies the start address of array in memory while index register stores relative position of each entry to start of array.

Some processors have dedicated one CPU register to function as index register. While in other processors (complex ones) have many registers each of them can act as index register.

10. Base Register Addressing mode:

The content of base register is added to the address part of the instruction. To obtain EA.

Similar to index addressing except register is called base register instead of index register

The difference can be seen as: base address holds the base address of arrays in memory while address part of instruction holds displacement relative to base register.

This addressing mode is used facilitate reallocation of programs in memory. When program and data are moved from a segment to another the relative position of data not changed while only its base address will be changed. The change in base register reflects start of new memory segment.

EXAMPLE:

Next figure shows a two word instruction at address 200 and 201 that "load to AC". The instruction has an address field occupying second word of value 500.

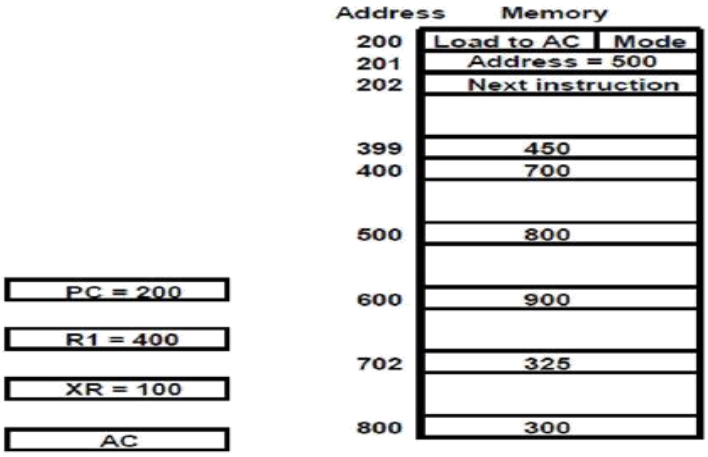
The first word specifies op code while second word specifies address of operand o
PC= 200

R1 = 400 (register)

XR = 100 (index register)

Mode field can specifies any mode mentioned earlier

1. In direct mode EA=500 and M[500] = 800. So AC = 800
2. In immediate mode EA=201. Second word of instruction is loaded to AC. So AC = 500
3. In Indirect mode M[500] = 800 which is EA. And M[800] = 300 be loaded into AC. So AC = 300.
4. In relative mode EA=500+202=702. M[702] = 325. So operand=325
5. In index mode EA=XR + 500 = 100 + 500 = 600. M[600] = 900. So AC = 900
6. In register mode operand is in R1. So 400 is loaded into AC. AC = 400
7. In register indirect mode EA=400. M[400] = 700. So AC = 700
8. In auto decrement mode AC= M[R1 – 1] = M[399] = 450



Addressing Mode	Effective Address			Content of AC
Direct address	500	/* AC ← (500)	*/	800
Immediate operand	-	/* AC ← 500	*/	500
Indirect address	800	/* AC ← ((500))	*/	300
Relative address	702	/* AC ← (PC+500)	*/	325
Indexed address	600	/* AC ← (RX+500)	*/	900
Register	-	/* AC ← R1	*/	400
Register indirect	400	/* AC ← (R1)	*/	700
Autoincrement	400	/* AC ← (R1)+	*/	700
Autodecrement	399	/* AC ← -(R)	*/	450

To illustrate how zero, one, two, and three address instruction differ, the next example will be introduced. If have next expression to be evaluated

1. **Three-Address Instructions**

Program to evaluate X = (A + B) * (C + D):

```
ADD R1, A, B        /* R1  M [A] + M[B]*/
ADD R2, C, D        /* R2  M[C]+M[D]*/
MUL X, R1, R2       /* M[X--]   R1 * R2*/
```

Results in short programs

Instruction becomes long (many bits)

2. Two-Address Instructions

Program to evaluate X = (A + B) * (C + D):

```
MOV R1, A           /* R1  M[A] */
ADD R1, B           /* R1  R1 + M[A] */
MOV R2, C
```

/* R2 M[C] */

```
ADD R2, D          /* R2  R2 + M[D] */
MUL R1, R2         /* R1  R1 * R2 */
MOV X, R1          /* M[X]  R1 */
```

3. **One-Address Instructions**

Use an implied AC register for all data manipulation

Program to evaluate X = (A + B) * (C + D):

```
LOAD A             /* AC  M[A] */
ADD B              /* AC  AC+ M[B] */
STORE T            /* M[T]  AC */
LOAD C             /* AC  M[C] */
ADD D              /* AC  AC + M[D]*/
MUL T              /* AC  AC * M[T]*/
STORE X            /* M[X]  AC */
```

4. **Zero-Address Instructions**

Can be found in a stack-organized computer

Program to evaluate X = (A + B) * (C + D):

```
PUSHA              /* TOS  A*/
PUSHB              /* TOS  B*/
ADD                /* TOS  (A + B)*/
PUSHC              /* TOS  C*/
PUSHD              /* TOS  D*/
ADD                /* TOS  (C + D)*/
MUL                /* TOS  (C + D) * (A + B) */
POPX               /* M[X]  TOS*/
```

Reduced Instruction Set Computers (RISC):

REDUCED INSTRUCTION SET COMPUTERS

In the late „70s and early "80s there was a reaction to the shortcomings of the CISC style of processors

Reduced Instruction Set Computers (RISC) were proposed as an alternative

The underlying idea behind RISC processors is to simplify the instruction set and reduce instruction execution time

RISC processors often feature:

- Few instructions
- Few addressing modes
- Only load and store instructions access memory
- All other operations are done using on-processor registers Fixe length instructions
- Single cycle execution of instructions
- The control unit is hardwired, not microprogrammed

Since all but the load and store instructions use only registers for operands, only a few addressing modes are needed

By having all instructions the same length, reading them in is easy and fast

The fetch and decode stages are simple, looking much more like Mano's Basic Computer than a CISC machine

The instruction and address formats are designed to be easy to decode

Unlike the variable length CISC instructions, the opcode and register fields of RISC instructions can be decoded simultaneously

The control logic of a RISC processor is designed to be simple and fast

The control logic is simple because of the small number of instructions and the simple addressing modes

The control logic is hardwired, rather than microprogrammed, because hardwired control is faster

BERKELY RISC I

32-bit integrated circuit CPU

32-bit address, 8-, 16-, 32-bit data

32-bit instruction format

Total 31 instructions

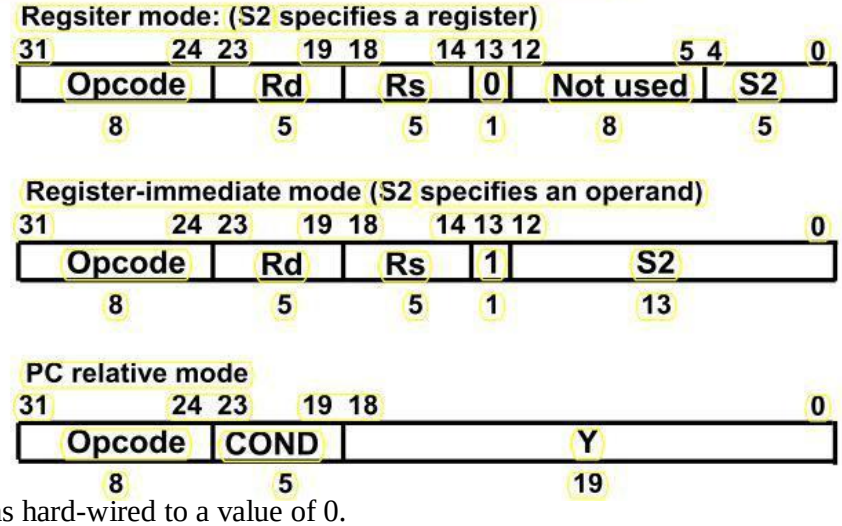
three addressing modes: register; immediate; PC relative addressing

138 registers

- global registers

- 8 windows of 32 registers each

Berkeley RISC I Instruction Formats



Register 0 was hard-wired to a value of 0.
There are eight memory access instructions

- Five load-from-memory instructions
- Three store-to-memory instructions.

INSTRUCTION SET OF BERKELEY RISC I

Opcode	Operands	Register Transfer	Description
Program control instructions			
JMP	COND,S2(Rs)	PC ← Rs + S2	Conditional jump
JMPR	COND,Y	PC ← PC + Y	Jump relative
CALL	Rd,S2(Rs)	Rd ← PC, PC ← Rs + S2 CWP ← CWP - 1	Call subroutine and change window
CALLR	Rd,Y	Rd ← PC, PC ← PC + Y CWP ← CWP - 1	Call relative and change window
RET	Rd,S2	PC ← Rd + S2 CWP ← CWP + 1	Return and change window
CALLINT	Rd	Rd ← PC,CWP ← CWP - 1	Call an interrupt pr.
RETINT	Rd,S2	PC ← Rd + S2 CWP ← CWP + 1	Return from interrupt pr.
GTLPC	Rd	Rd ← PC	Get last PC

INSTRUCTION SET OF BERKELEY RISC I

Advantages of RISC:

VLSI Realization
Computing Speed
Design Costs and Reliability
High Level Language Support

Design Costs and Reliability:

Shorter time to design (reduction in the overall design cost and reduces the problem that the end product will be obsolete by the time the design is completed)
Simpler, smaller control unit (higher reliability)
Simple instruction format (of fixed length) (ease of virtual memory management)

High Level Language Support:

A single choice of instruction (shorter, simpler compiler)
A large number of CPU registers (more efficient code)
Register window (Direct support of HLL)
Reduced burden on compiler writer

Instruction Codes

In this chapter we introduce a basic computer and show how its operations can be specified with RTL statements.

Every different processor type has its own design (different registers, buses, micro-operations, machine instructions, etc)

Basic Computer Organization

Modern processor is a very complex device. It contains:

- Many registers
- Multiple arithmetic units, for both integer and floating point calculations
- The ability to pipeline several consecutive instructions to speed execution

However, to understand how processors work, we will start with a simplified processor mode. This is similar to what real processors were like ~35 years ago

M. Morris Mano introduces a simple processor model he calls the Basic Computer.

We will use this to introduce processor organization and the relationship of the RTL model to the higher level computer processor

The internal organization of a digital system is defined by the sequence of micro-operations it performs on data stored in its registers. By executing several micro-operations in specified sequence, then a computer instruction can be executed.

Program: A set of instructions that specifies operation, operands, and sequence of processing has to occur

- The instructions of a program, along with any needed data are stored in memory.
- The CPU reads the next instruction from memory.
- It is placed in an Instruction Register(IR)
- Control circuitry in control unit then translates the instruction into the sequence of micro-operations necessary to implement it

Computer Instruction: A binary code that specifies a sequence of micro-operations for the computer. Every computer has its own unique instruction set

Instruction code: is a group of bits instructing the computer to perform a specific operation. It is divided into parts, each with particular meaning.

Operation code: the most important part of instruction code and it defines the type operation like add, subtract, multiply, shift, and complement.

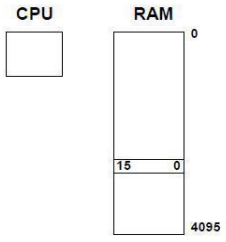
Stored Program Organization

The Basic Computer has two components, a processor and memory

The memory has 4096 words in it

- 4096 = 212, so it takes 12 bits to select a word in memory

Each word is 16 bits long



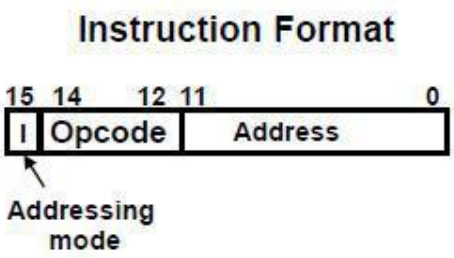
A computer instruction is often divided into two parts

1. An op-code (Operation Code) that specifies the operation for that instruction
2. An address that specifies the registers and/or locations in memory to use for that operation

In the Basic Computer, since the memory contains 4096 (= 212) words, we needs 12 bit to specify which memory address this instruction will use

In the Basic Computer, bit 15 of the instruction specifies the addressing mode (0: direct addressing, 1: indirect addressing)

Since the memory words, and hence the instructions, are 16 bits long, that leaves 3 bits for the instruction's op-code



Accumulator Register (AC): exist in single register processors (AC) and all operations are performed with memory operand and this register.

The previous figure shows an example of addressing mode. In location 22 there is an ADD instruction that adds the AC with operand in location 457 as an indication for direct addressing mode as I=0.

On the other hand, the second part of the figure shows in location 35 and ADD instruction between AC and the address of operand found in location 300. Location 300 contains the operand address of 1350. In 1350 the operand will be read and added to the AC register, as I=1 shows.

Computer Registers:

The purpose this register, the **Program Counter (PC),** is to hold the memory address of the next instruction to be executed. And since the memory in the Basic Computer only has 4096 locations, the PC only needs to be in 12 bits.

- Program Counter(***PC***) :
    - hold the address of the next instruction to be read from memory after the current instruction is executed

- Instruction words are read and executed in sequence unless a branch instruction is encountered
- A branch instruction calls for a transfer to a nonconsecutive instruction in the program
- The address part of a branch instruction is transferred to PC to become the address of the next instruction
- To read instruction, memory read cycle is initiated, and PC is incremented by one(next instruction fetch)

In a direct or indirect addressing, the processor needs to keep track of what locations in memory it is addressing: The **Address Register (AR)** is used for this type of application.
The AR is a 12 bit register in the Basic Computer as it always holds the address of next access of memory. This AR register is always connected to address pins of memory unit.

When an operand is found, using either direct or indirect addressing, it is placed in **the Data Register (DR).** The processor then uses this value as data for its operation. DR is connected to data pins of memory unit.

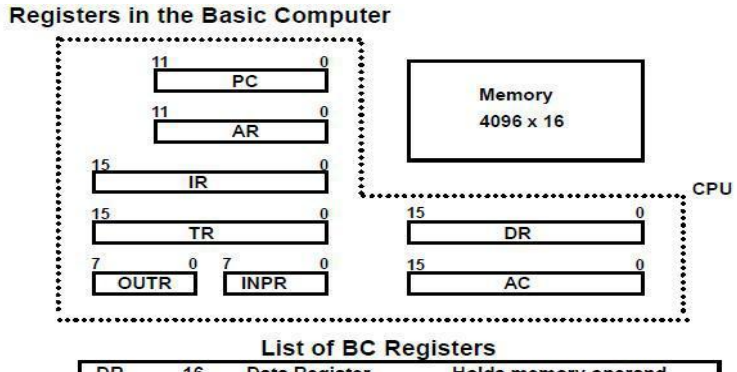
The Basic Computer has a single general purpose register–the **Accumulator (AC).** The significance of a general purpose register is that it can be referred to in instructions. E.g. load AC with the contents of a specific memory location; store the contents of AC into a specified memory location. So AC is always one side in any data computation.

Often a processor will need a scratch register to store intermediate results or other temporary data; in the Basic Computer this is the **Temporary Register (TR).**

For instruction reading, a PC is used to hold address of next instruction to be fetched and executed. First it's the PC valued is copied into AR to start an instruction reading cycle, then the 16 bit instruction is fetched and placed in **Instruction Register (IR).**

The Basic Computer uses a very simple model of input/output (I/O) operations. Input devices are considered to send 8 bits of character data to the processor. And the processor can send 8 bits of character data to output devices.

The **Input Register (INPR)** holds an 8 bit character got from an input device; whereas; the **Output Register (OUTR)** holds an 8 bit character to be send to an output device.



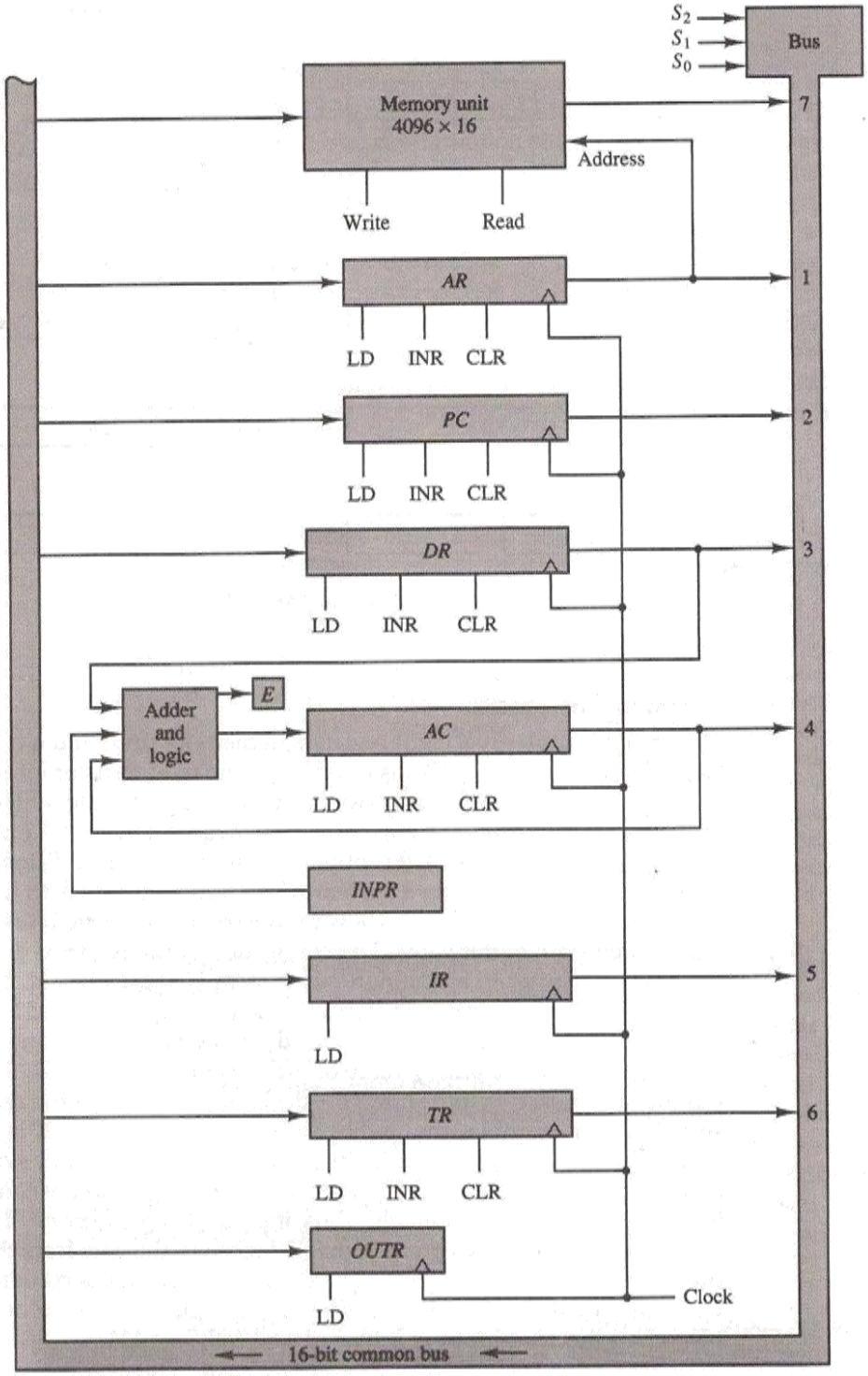
DR	16	Data Register	Holds memory operand
AR	12	Address Register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction Register	Holds instruction code
PC	12	Program Counter	Holds address of instruction
TR	16	Temporary Register	Holds temporary data
INPR	8	Input Register	Holds input character
OUTR	8	Output Register	Holds output character

Bus:

- A more efficient scheme for transferring information in a system with many registers is to use a common bus.
- The basic computer has eight registers, a memory unit, and a control unit.
- Paths must be provided to transfer information from one register to another and between memory and registers
- A more efficient scheme for transferring information in a system with many registers is to use a common bus.
- The registers in the Basic Computer are connected using a bus
- This gives a savings in circuitry over complete connections between registers
- Three control lines, S_2, S_1, and S_0 control which register the bus selects as its input
- Either one of the registers will have its load signal activated, or the memory will have its read signal activated

  Will determine where the data from the bus gets loaded
- The 12-bit registers, AR and PC, have 0's loaded onto the bus in the high order 4 bit positions
- When the 8-bit register OUTR is loaded from the bus, the data comes from the low order 8 bits on the bus.

Common Bus System :



S_2	S_1	S_0	Register
0	0	0	x
0	0	1	AR
0	1	0	PC
0	1	1	DR
1	0	0	AC
1	0	1	IR
1	1	0	TR
1	1	1	Memory

A path needed to transfer data between 8 registers beside memory unit and registers. So a common bus will be the answer for that problem. The next figure shows the answer consisting of a multiplexer or 3 state buffers with decoder. This gives a savings in circuitry over complete connections between registers.

Three control lines, S2, S1, and S0 control which register the bus selects as its input and so the selected register will issue its output to the bus.

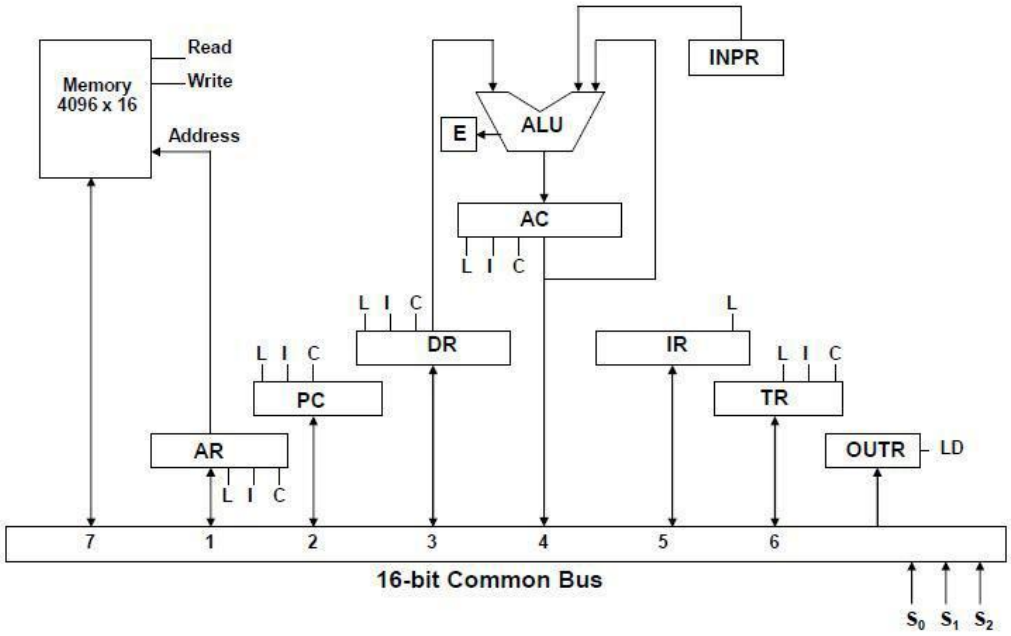
The lines from the common bus are connected to the input of each register. Either one of the registers will have its load signal activated, or the memory will have its read signal activated. And this will determine where the data from the bus gets loaded to during next clock transition.

The memory will put its content to the bus when S2S1S0 =111 and its read control signal is activated. In the same manner, the memory will save the content of the bus when its write control signal is activated. AR register is always used to hold address of data accessed from memory.

We have in this basic computer 4 registers of 16-bit each, DR, AC, IR, and TR. Also we have two registers PC and AR are 12 bits each. The 12-bit registers, AR and PC, have 0's loaded onto the bus in the high order 4 bit positions. OUTR and INPR are 8-bit each and are connected to the lower 8 bits of the bus.

Five registers have 3 control inputs, LD, INC, CLR; those are AC, AR, TR, PC, and DR. 2 registers, OUTR, and IR will only have a LD input.

The data coming to the AC register comes from ALU unit which accepts operands from AC register, INPR register, or DR register.



Any and only one source of those register can be selected to apply its content to the bus and during the same clock cycle the bus content could be directed to one or many destinations of registers or memory unit. For example we can do the next micro-operation:

DR AC and AC DR

S2S1S0 = 100

LD of DR is enabled

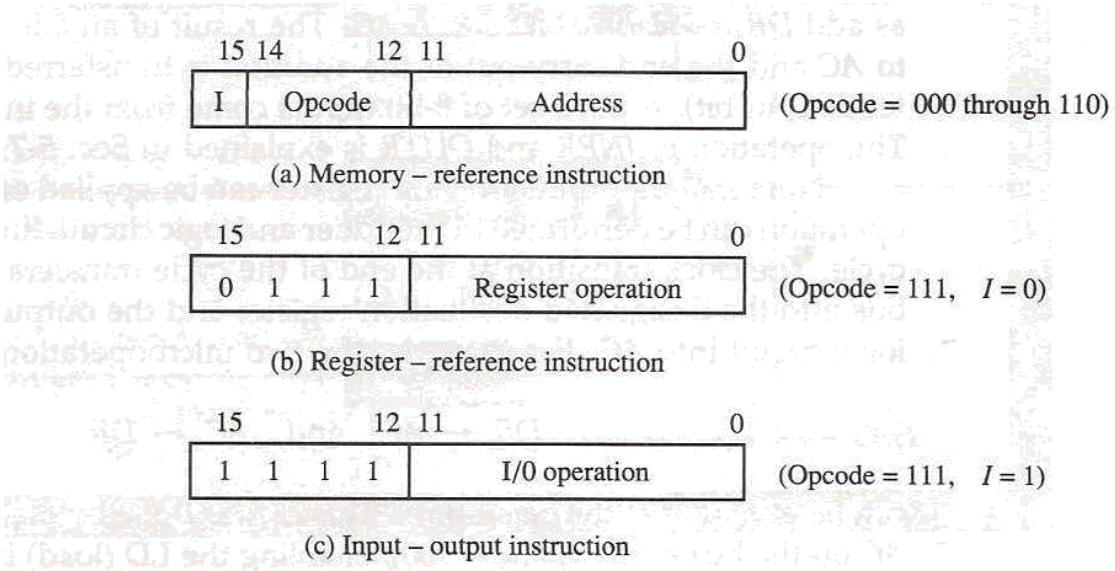
Transferring DR through ALU to AC

LD of AC is enabled

Computer Instructions

The Basic Computer has 3 instruction code formats as shown in next figure. Each format has 16 bits.

- The op-code of the instruction contains 3 bits and the meaning of the remaining 13 bits depends on the operation code encountered.
- Memory reference instruction uses 12 bits to specify the operand address and one bit for indirect address.
- The register reference instruction are recognized by op-code 111 with 0 in left most bit (Bit 15) of the instruction. The 12 bits are used in to specify the operation done with AC register.
- Input-Output instruction is recognized by op-code 111 and with 1 in bit 15. The remaining 12 bits are used to specify type of Input-Output instruction type.



- Only 3 bits of the instruction are used for the operation code, op-code, and since register reference instructions and IO instructions use the other unneeded 12 bits, so the total number of instruction coded in this computer is 25.

Symbol	Hexadecimal code $I = 0$	Hexadecimal code $I = 1$	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instruction if AC positive
SNA	7008		Skip next instruction if AC negative
SZA	7004		Skip next instruction if AC zero
SZE	7002		Skip next instruction if E is 0
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

Timing & Control

The timing for all registers is controlled by a master clock. The clock pulses generated do not change the state of a register unless it is enabled by a control signal.

Control unit (CU) of a processor translates from machine instructions to the control signals for the micro-operations that implement them. The control signals are generated in the control unit and provides control inputs to

All register

Multiplexers

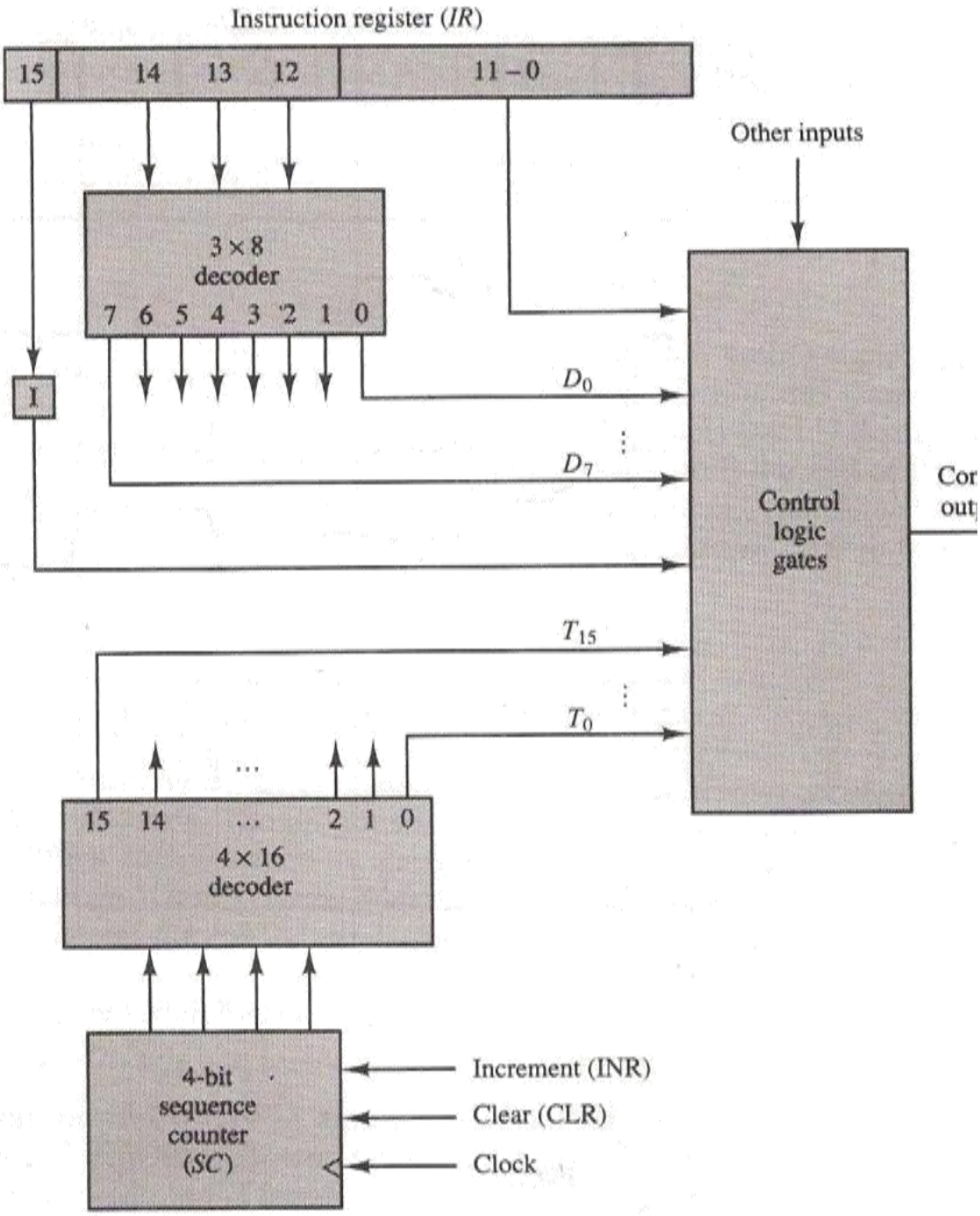
Common bus

And micro-operation indicators

Control units are implemented in one of two ways:

- Hardwired Control
    - o CU is made up of sequential and combinational circuits to generate the control signals
- Micro-programmed Control
    - o A control memory on the processor contains micro-programs that activate the necessary control signals
- We will consider a hardwired implementation of the control unit for the Basic Computer

The next figure shows a block diagram of control unit of the basic computer



It consists of 2 decoders, a sequence counter, and a number control logic gates.

- The instruction register holds the instruction fetched from memory. It is divided into 3 parts: I bit, op-code, and 12 bits.

  o Op-codes is divided by 3 by 8 decoder into 8 different outputs; D0 to D7 o

  Bit 15 is transferred to I flip flop

o Bit 0 to Bit 11 (B0 to B11) are applied to control logic gates.

- The 4-bit sequence counter is connected to 4 by 16 decoder giving control inputs T0 to T15.

  o Used to synchronize action with those 16 different time intervals.

  o Sequence counter can be incremented and cleared synchronously.

  o Most of the time it is incremented to generate sequence of timing signals

  o Once in a while it is cleared by specific condition causing next timing sequence to go back to T0.

Example on Timing Control:

- Consider a case where SC is incremented to provide timing sequence T0, T1, T2, T3, and T4, then T0 again. At time T4 Sc is cleared based on a condition D3 is true. Expressed in symbolic RTL form:

  D3T4: SC 0

  The next figure shows its timing diagram. When D3T4 is true then at first positive clock transition SC is cleared.

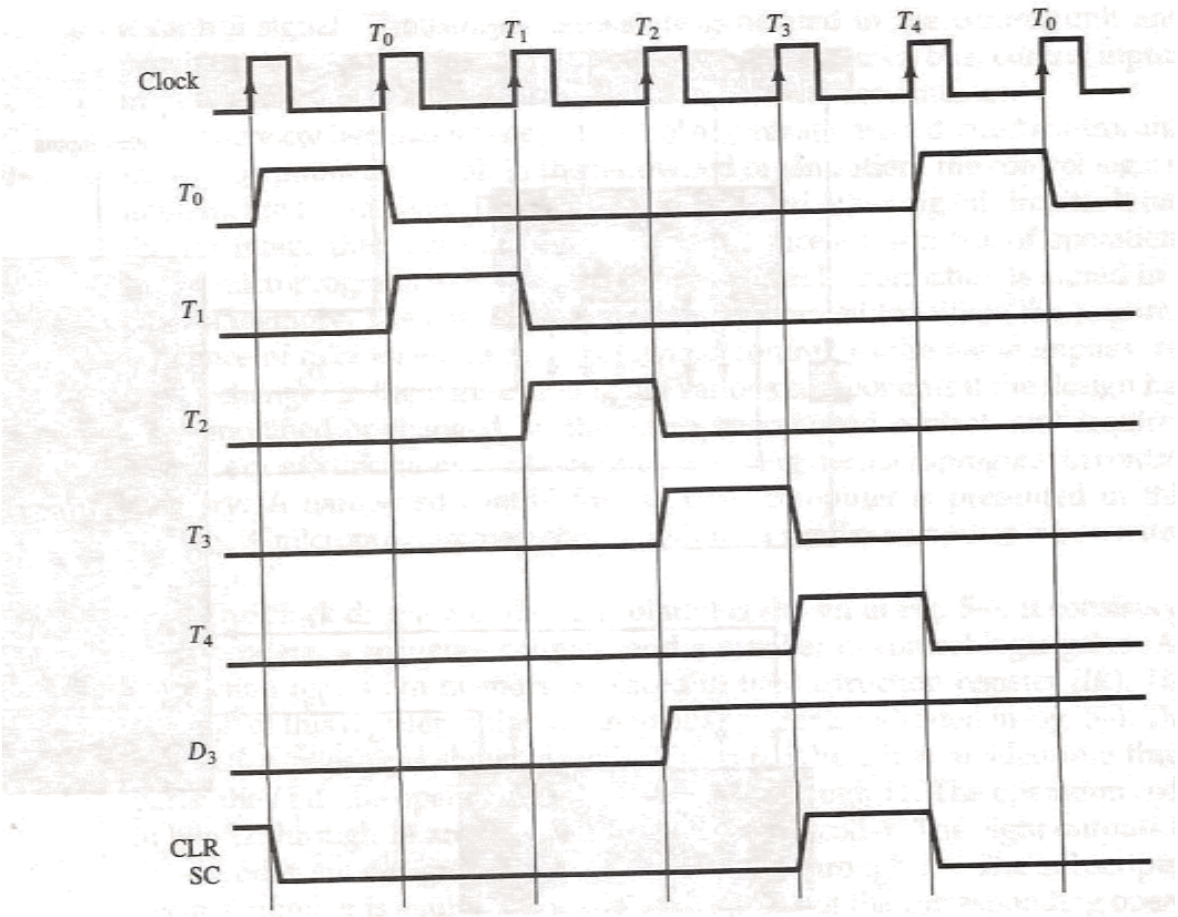
I f SC is not cleared then it will continue its counting from T5 to T15 then it rolls over to T0 again.

- For a memory read operation, it must be clear that between 2 rising edge of the clock, the data should be read and applied to the bus, so that at next rising edge the data can be saved in destination register.

- The next symbolic RTL

  T0: AR PC

  That specifies transfer of data from PC to AR register in one clock pulse T0. The content PC is put on the bus (S2S1S0=010) and LD of AR register is enabled during T0 cycle only



.

Instruction Cycle

In Basic Computer, a machine instruction is executed in the following cycle:

- Fetch an instruction from memory
- Decode the instruction
- Read the effective address from memory if the instruction has an indirect address
- Execute the instruction

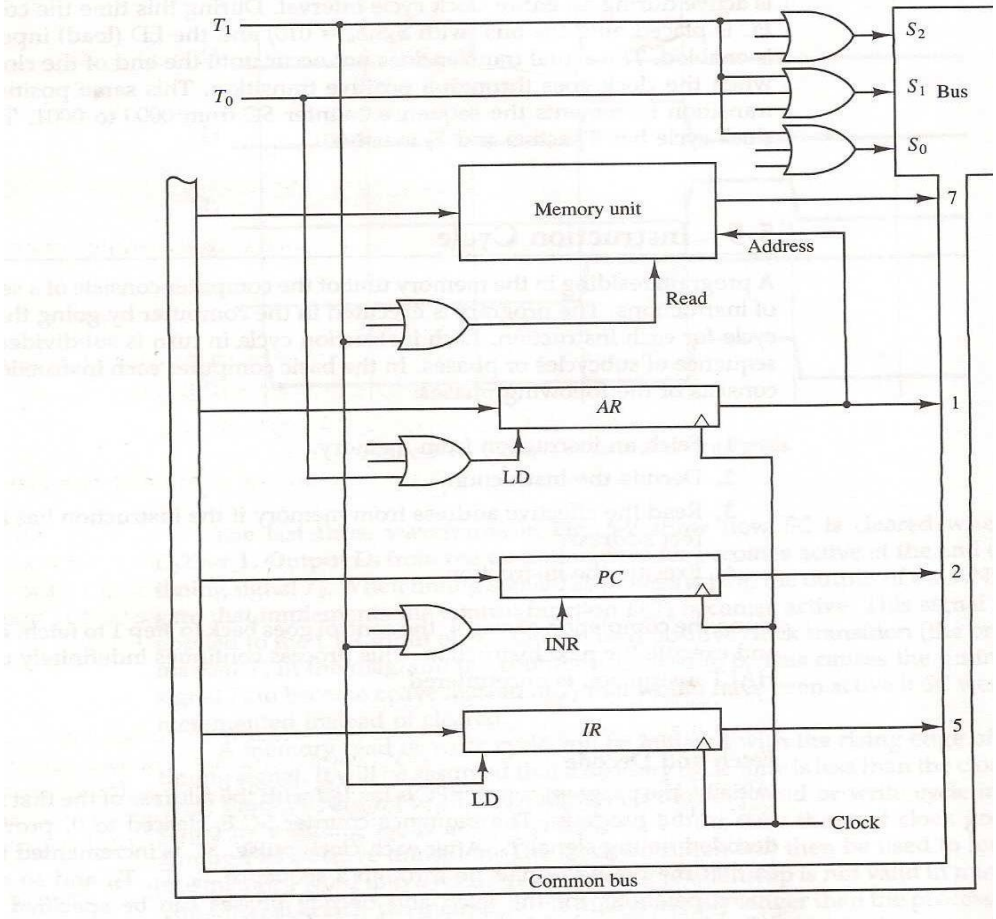
After an instruction is executed, the cycle starts again at step 1, for the next instruction

Fetch and Decode:

Initially the PC is loaded with address with first instruction in the program then SC is cleared giving time instance T0. After each clock pulse SC is incremented resulting of T1, T2, and so on. The fetch and decode phase micro-operations would be:

T_0: $AR \leftarrow PC$
T_1: $IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
T_2: $D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14)$, $AR \leftarrow IR(0\text{–}11)$, $I \leftarrow IR(15)$

Next figure shows a portion of bus system and shows hardware implementation of micro-operations taking place during T0 and T1



1.. During T0

- o S2S1S0=010 which means PC apply its output to the bus
- o AR register LD = 1 which means what in the bus goes into AR register

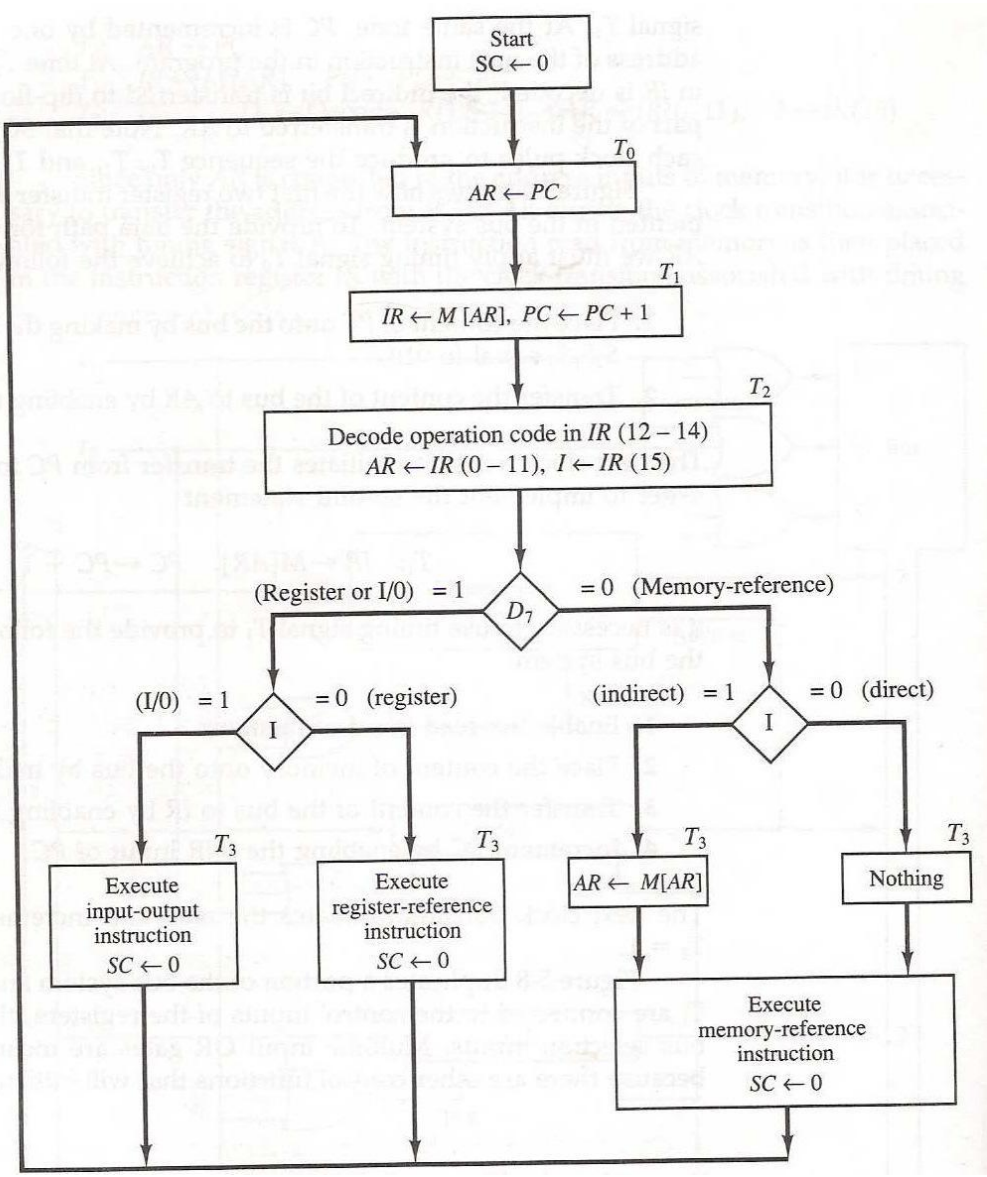
2. During T1

- o Memory RD=1 which means memory unit will get its data out as AR register indicates.
- o S2S1S0=111 which means what is read from memory goes to the bus

o IR register LD = 1 which means what on the bus will go into IR register o PC register INR = 1 which means increment PC register.

Flow chart to Determine the Type of Instruction (for memory reference / register reference / IO reference)

During T3 the type of instruction will be decoded in steps as shown in next figure. It shows how control determines the instruction type after the decoding.



- If D7=1 then the instruction must be register reference or Input-Output instruction
    - o D7.I' = Register reference Instruction o
      D7.I = IO Instruction
- If D7=0 then it may be 000 to 110 values = D0 to D6 which specifies memory reference instructions.
    - o If I=1 then it will be indirect memory reference instruction otherwise if I=0 then it will be direct memory reference instruction.
    - o D7'.I' = memory reference direct addressing mode o
      D7'.I = memory reference Indirect addressing mode
      AR M[AR]
- During T3 one of 4 different paths
    - o D7'IT3 = AR M[AR]
    - o D7'I'T3 = Nothing
    - o D7I'T3 = execute register reference instruction o
      D7IT3 = execute input-output instruction
- In timing T4 memory reference instruction will execute
- SC is either incremented to enable computer to go to next timing sequence, or set to zero to indicate the termination of instruction execution and start new instruction fetch cycle.

Register Reference Instructions

- Register reference instruction will be recognized by control unit if o
  D7= 1 and
  o I = 0
- Register Ref. Instr. is specified in b0~ b11of IR,
    - o Bi= IR(i) , i=0,1,2,...,11 is the bit which indicate each instruction.
    - o Its condition will be summarized by r = D7.I'.T3
    - o One in each bit from 0 to 11 specifies a different register reference instruction.
    - o Execution of register reference starts with timing signal T3 and completed here. Also Sc is cleared to indicate the end of execution and return to fetch new instruction with T0.
- Next figure summarizes register reference instructions

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0–11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow \text{shr } AC, AC(15) \leftarrow E, E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow \text{shl } AC, AC(0) \leftarrow E, E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

Remarks on Register Reference Instructions:

- Note that the first seven instructions will be carried on accumulator or carry bit, E bit.

- Note that too, the next 4 skip instructions will add one to PC register only if their condition is met. Those conditions will be

    - Accumulator is positive, A15=0
    - Accumulator is negative, A15=1
    - Accumulator is zero, A0 to A15 are zero.
    - E is 0, carry bit is zero.

REGISTER TRANSFER AND MICROOPERATIONS

Register Transfer Language

A digital system is an interconnection of digital hardware modules that accomplish specific information-processing task.

Digital systems are built from modules that are built from components such as registers, decoders, arithmetic elements and control logic. Those modules are connected with control and data paths.

Digital modules are best defined by registers they contain and operations they perform on stored data.

Micro operations: the operations executed on data stored in registers. An elementary operation performed in information stored in register(s). Examples of micro operations: shift, count, clear, increment, and load

But also it is interesting to know that the internal hardware organization of a digital computer is best defined by:

Registers it contains and their functions

Sequence of micro operations performed on data inside registers

Control that ignites the sequence of micro operations

It is time now to agree on a terminology to describe the sequence of transfer between registers and arithmetic and logical operations associated with those transfers.

Register transfer language: is symbolic notations used to describe micro operation transfer among registers. RTL is a system for expressing in symbolic form the micro operation sequences among registers in a digital module

Register Transfer

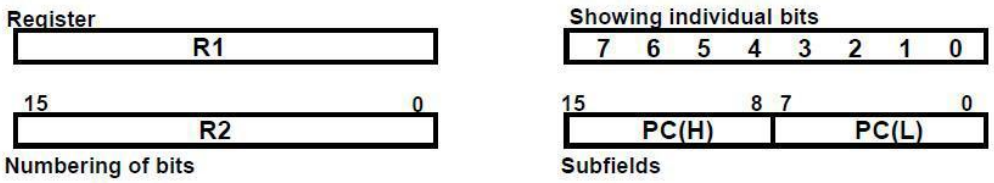
Designation of Registers:

Registers are designated by capital letters; sometimes followed by numbers to denote the function of a register. Examples will as:

MAR	for Memory Address Register
PC	for Program Counter
IR	for Instruction register

The individual flip flops in n-bit register is numbered from 0 in right most to n-1 in left most

A register can be viewed as a single entity or may also be represented showing the bits of data they contain. Registers can be designated by a whole register, portion of a register, or a bit of a register. Registers and their contents can be viewed and represented in various ways such as shown in next figure:



Register transfer:

Register Transfer is defined as copying the content of one register to another. For register transfers, the data transfer from one register to another is designated in symbolic form by replacement operator

R2R1

Note:

- In this case the contents of register R2 are copied (loaded) into register R1
- 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

That register transfer also implies that:

- The data lines extend from the source register (R1) to the destination register (R2) with lines equal the bit numbers of R1 and R2.
- Parallel load occurs in the destination register (R2)
- Control lines are needed to perform this action

Control Function:

We need the transfer to happen under a certain condition by means which looks if-then statement. In digital systems, this is often done via a control signal, called a control function. If the signal is "1" then action will take place. See next example for control statement P. P also could be a combination of Boolean variables which yields a single Boolean output.

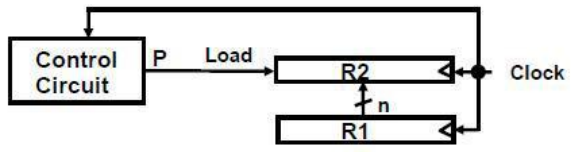
P:R2R1

Which means "if P = 1, then load the contents of register R1 into register R2", i.e., if (P =1) then (R2 R1). Next figure shows the hardware implementation for implementing control functions.

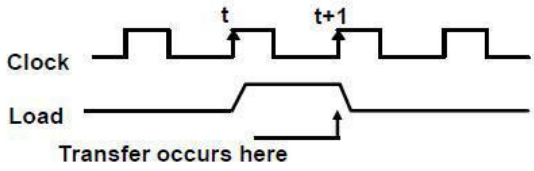
Hardware Implementation for control function:

Look at next diagram which shows R1 transfer to R2. You will realize that the n outputs of register R1is connected to n input of register R2. Register R2 has a load control activated by P control function and the whole operation is synchronized with the central clock. The rising edge of the CLK input triggers activates P at t time and at t+1 time the transfer takes place.

Block diagram



Timing diagram



Also we assume that here the registers are comprised of DFF that acts on rising edge clocks.

Simultaneous Operations:

If cases where two or more operations are to occur simultaneously, they are separated with commas as shown next:

P: R3 R5, MAR 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

The basic symbols for register transfer is shown in next table and that summarizes the topic

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

Bus and Memory Transfer

There is a problem if we need to move data from and to multiple registers. The number of wires will be so large if separate lines are used to connect all registers with each other.

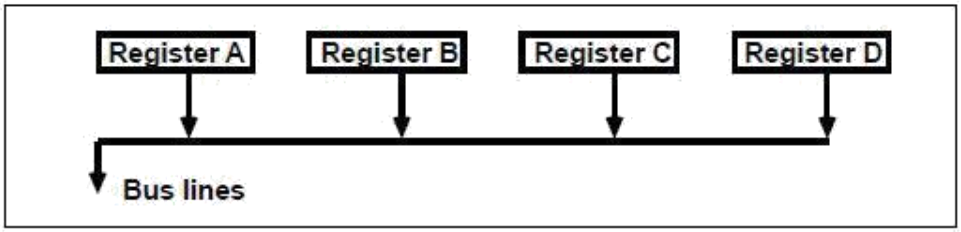
To completely connect n registers we need n(n-1) lines. So the cost is in order of O(n2). This is not a realistic approach to be used in a large digital system. The solution is to use a common "Bus".

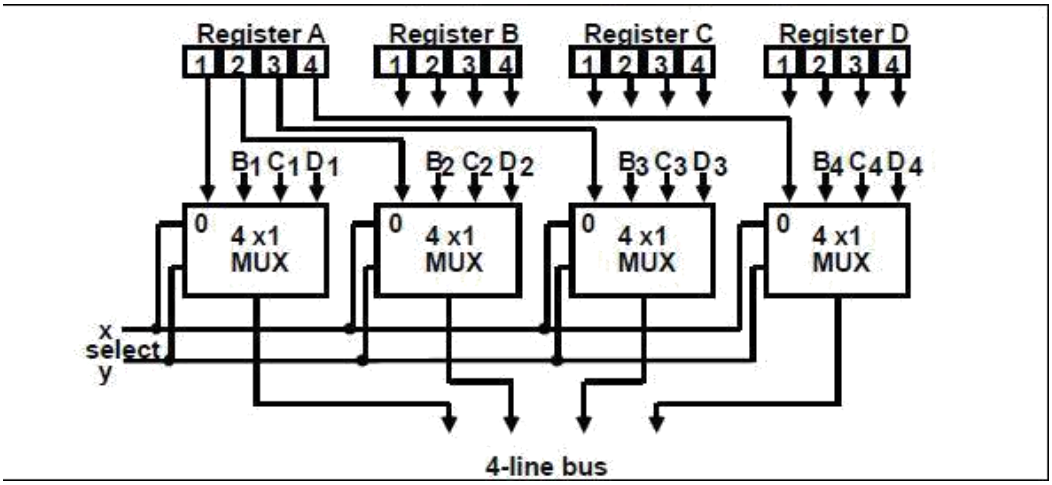
Instead, take a different approach; have one centralized set of circuits for data transfer the "bus". Also have control circuits to select which register is the source, and which is the destination.

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

One way of constructing a bus is by using multiplexers. The next diagram shows how this works. The next figure shows how to implement dart transfer from register to the bus.

BUSR

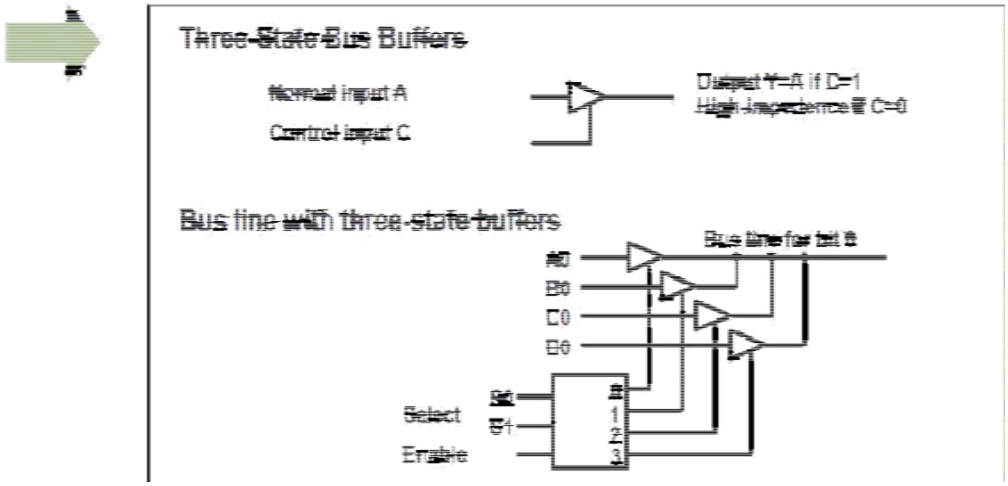




In general, the bus system will multiplex k registers of n bits each to produce an n-line common bus.

- The number of multiplexers need is n
- The size of each multiplexer is k X 1

Another way of constructing a bus is by using buffers or 3-state gates. The next two figures show the buffers in use for constructing a bus system.



The first one shows the graphical symbol for 3-state buffer while the next one shows bus system using 4 by one selectors of buffers.

Bus Transfer in RTL:

The transfer of information from a bus into one of many destination registers can be accomplished by connecting bus lines to the inputs of all registers and activating load control of selected destination. The symbolic statement for a bus transfer may mention the bus or may be implied in the statement.

R2R1

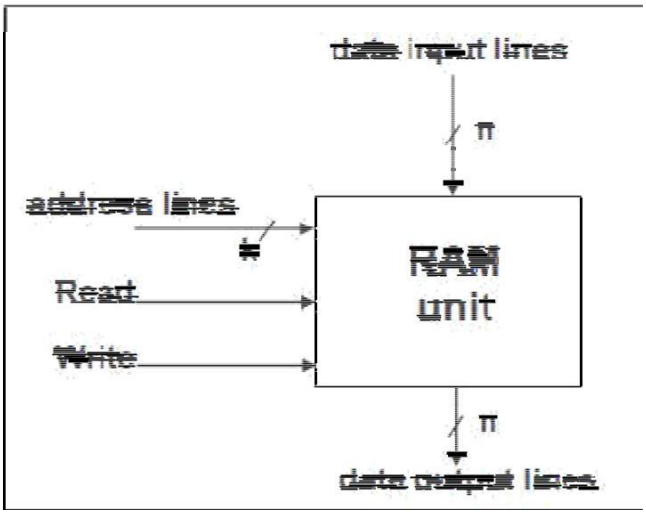
OR

BUSR1, R2BUS

Memory Transfer:

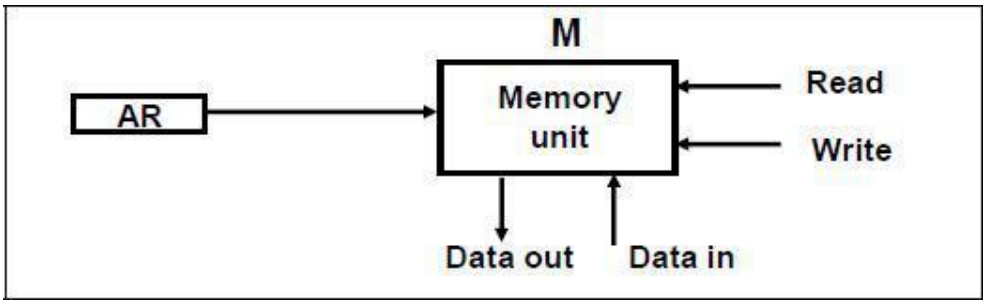
Memory (RAM) can be thought as a sequential circuits containing some number of registers. These registers hold the words of memory. Each of the r registers is indicated by an address. These addresses range from 0 to r-1. Each register (word) can hold n bits of data. Now assume the RAM contains r = 2k words. It needs the following

n data input lines
data output lines
k address lines
A Read control line
A Write control line



The memory can be viewed at the register level as a device, M. And since it contains multiple locations, then we must specify which address in memory we will be using.

This is done by indexing memory references. Memory is usually accessed in computer systems by putting the desired address in a special register, the Memory Address Register (MAR, or AR). And when memory is accessed, the contents of the MAR get sent to the memory unit's address lines.



For Read Operation: when address of required location is transferred into address register AR then the content is loaded into data register DR.

DRM[AR]

For Write operation: the content of data register DR is transferred into memory location addressed by address register AR.

M[AR]DR

Arithmetic Micro operation

A micro operation is an elementary operation performed with data stored in register. They are classified into:

- Register transfer micro operation
- Arithmetic micro operation
- Logic micro operation
- Shift micro operation

Basic arithmetic micro operations are:

- Addition
- Subtraction
- Increment
- Decrement
- Arithmetic Shift

Short look on different arithmetic micro operations:

The Add micro operation is specified as:

R3R1+R2

And it means add content of R1 to R2 and store result of addition in R3. Usually it is implemented using hardware full adders.

The Sub addition is usually implemented using complementation and addition

R3R1+R2+1(R1-R2)

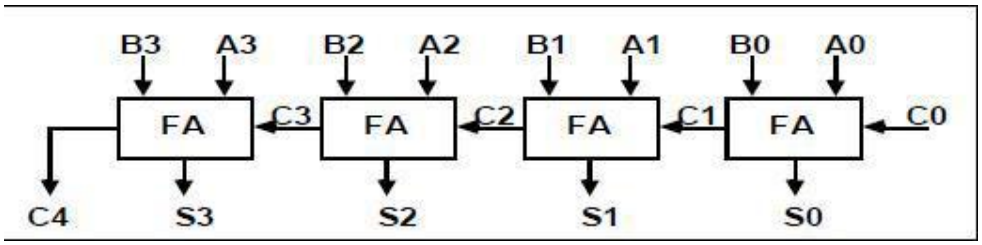
And it means subtract R2 from R1 by adding the complement of R2 plus 1 to R1. Usually it is implemented using a full adder a complement circuit.

Nevertheless, Increment and decrement are implemented using Up and Down Counter. And finally multiplication and division are implemented using sequence of additions and subtractions respectively.

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

Binary Adder:

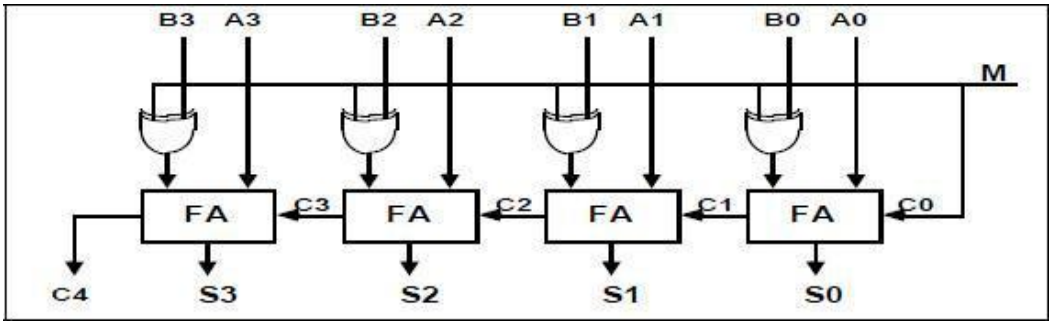
To implement binary adder we need registers that holds data and a full adder that performs arithmetic operation between 2 bits and previous carry. Binary adders are constructed from full adders connected in cascade. N-bit binary adder circuir need n number of full adders.



Binary Adder-Subtractor:

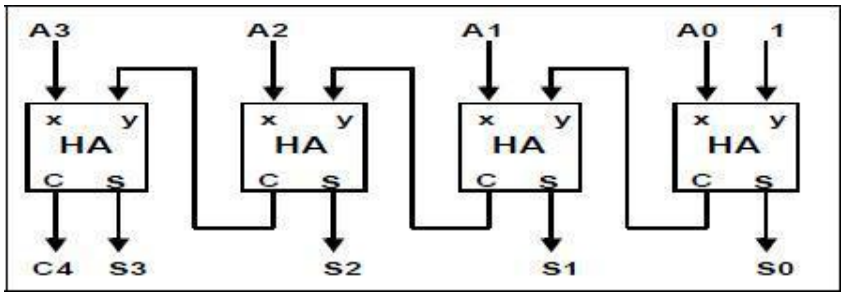
Subtraction of A – B can be done by taking 2's complement of B and added to A. The 2's complement can be done by taking 1's complement then adding "1" to the result. And finally the 1's complement is the binary inversion.

The addition and subtraction operations can be combined into one common circuit by including ExOR with each full adder. By looking at next drawing you will notice that M control addition or subtraction operations. If M=0 then it is an adder and if M=1 then it is a subtractor



Binary Incrementor:

The binary incrementor is defined as it always adds one to the number in a register. The incrementor can be implemented in one way by a counter. When clock transition arrives the count is incremented. But in another way the incrementor can be done using half adders. The next drawing shows in hardware an incrementor of 4 bits. It can be extended to n bits easily. As you notice the least significant adder always have one of its input as "1" while its carry is cascaded to other half adders.



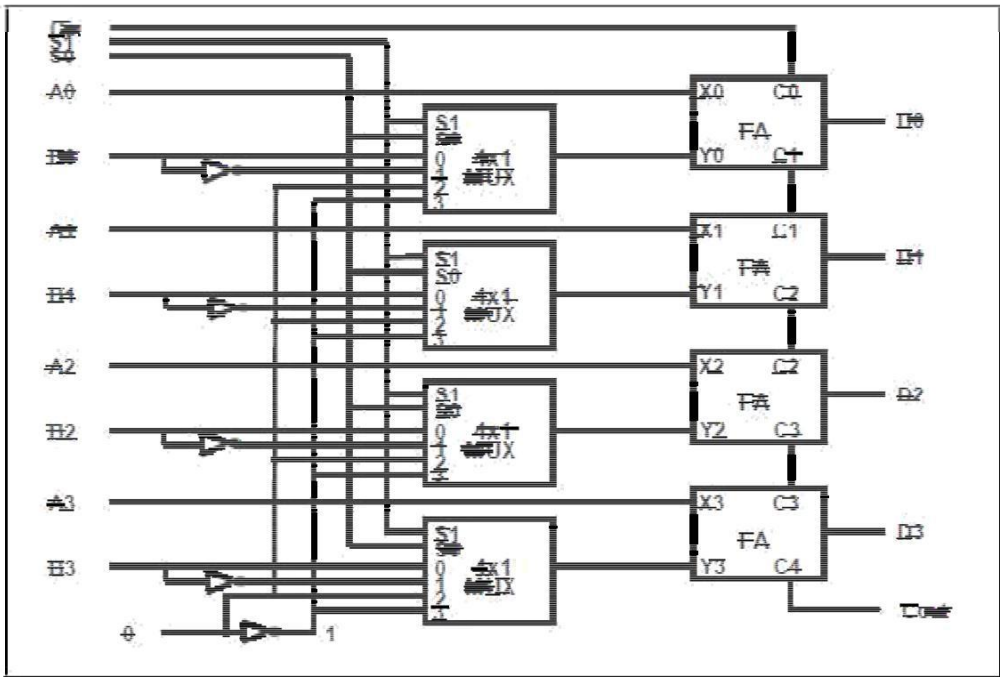
Arithmetic Circuit (Hardware Circuits)

The arithmetic micro operations listed in table 4-4 can be implemented in one composite arithmetic circuit. This circuit comprised of full adders and multiplexers. The multiplexer controls which data is fed into Y input of the adder.

The output of the binary adder is computed from

$$D = A + Y + C_{in}$$

The Y input can have one of 4 different values: B, B', always "1", or always "0". The next table shows how this can be implemented.



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 Micro operation (Hardware Circuits)

Logic micro operation specify binary operations on the strings of bits in registers. Logic micro operations are bit-wise operations, i.e., they work on the individual bits of data.

Those could be useful for bit manipulations on binary data and also 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. However, most systems only implement four of these:

AND ($\wedge$), OR ($\vee$), XOR ($\oplus$), Complement/NOT

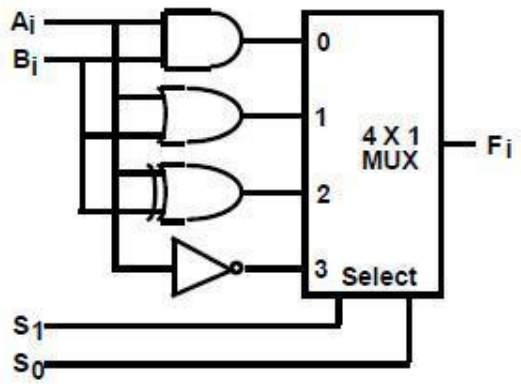
The others can be created from combination of these. List of Logic Microoperations-16

different logic operations with 2 binary variables are shown next.

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

The hardware implementation of logic micro operation requires the insertion of the most important gates like AND, OR, EXOR, and NOT for each bit or pair of bits in the registers.

The next figure shows one stage of a circuit that generates the four basic logic micro operations. It consists of four gates and a multiplexer. The two selection lines of the multiplexer selects one of the four logic operations available at one time. The circuit shows one stage for bit "i" but for logic circuit of n bits the circuit should be repeated n times but with one remark; the selection pins will be shared with all stages.



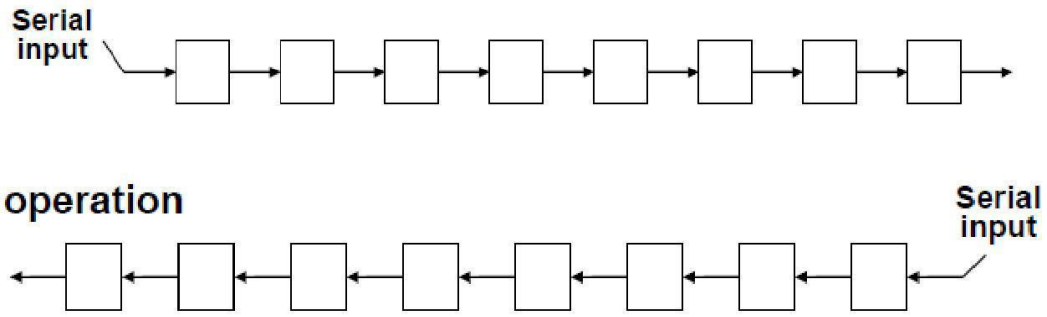
Function table

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

Shift Micro operations (Hardware Circuits)

Shift micro-operations are used for serial transfer of data beside they are used in conjunction with arithmetic, logic, and other data processing operations. There are 3 types of shift micro operations. What differentiates them is the information that goes into the serial input:

- Logical shift
- Circular shift
- Arithmetic shift



Logical Shift:

Logical shift is one that transfers 0 through the serial input. In a Register Transfer Language, the following notation is used

- shl for a logical shif teft
- shr for a logical shift right

Examples:

- R2SHR R2

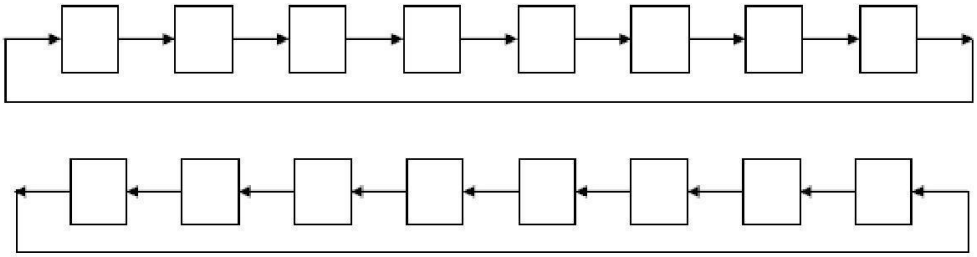
- R3SHR R3

Circular Shift:

The circular shift rotates of the register around the two ends without loss of information. This is accomplished by connecting the two ends of the shift register to each other. the following notation is used

- cil for a circular shift left
- cir for a circular shift right

The next two figures shows the circular shift right and left respectively.



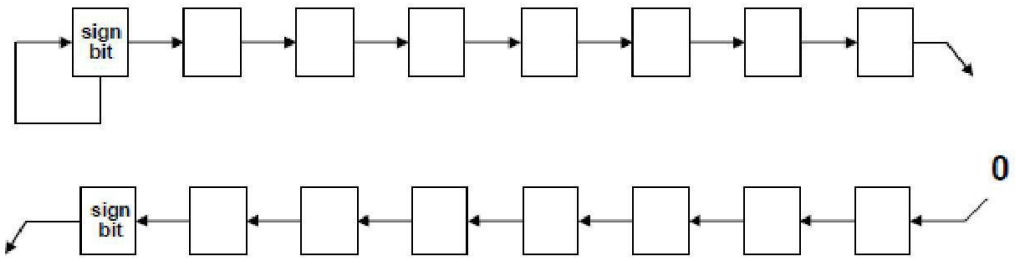
Examples:

- R2 cir R2
- R3 cil R3

Arithmetic Shift:

Arithmetic shift is a micro-operation that shifts a signed binary number to the left or right. Arithmetic shift must leave sign bit unchanged.

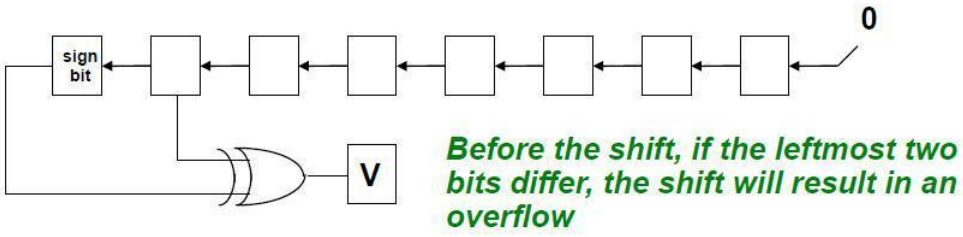
Note that the arithmetic shift right is considered divide by 2 and left shift is considered multiply by 2. The next two figures show the arithmetic shift right and left respectively.



- Arithmetic shifts must leave the sign bit unchanged just to preserve the

sign of the resulted number. If that case happened then it will be an overflow.

- An overflow flip flop will be used to detect arithmetic shift left overflow as shown in next figure.



In a RTL, the following notation is used for arithmetic shifts:

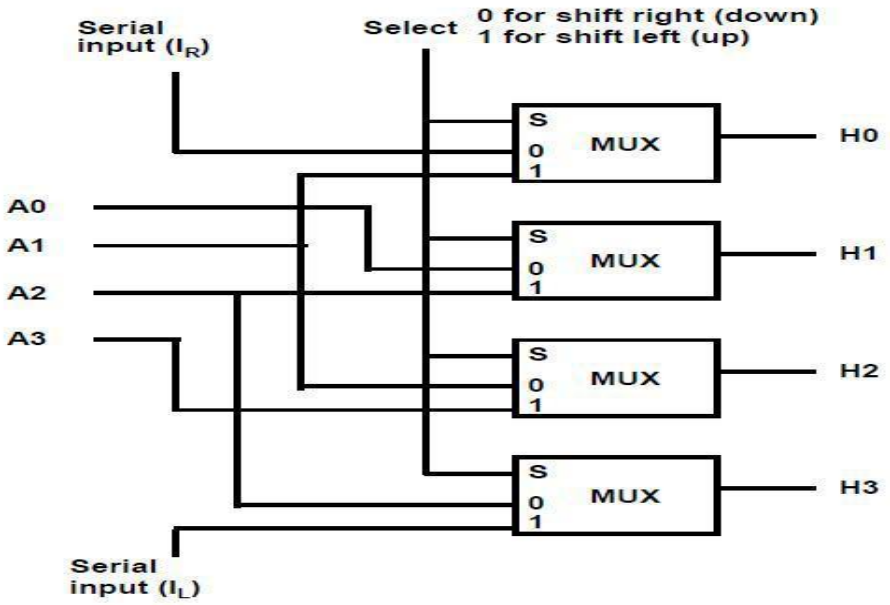
- ashl for an arithmetic shift left
- ashr for an arithmetic shift right

Examples:

- R2 ashr R2
- R3 ashl R3

Hardware Implementation:

One possible for a shift unit would be bidirectional shift register with parallel load as shown in chapter 2. But another solution can be constructed from multiplexers as shown in next figure.



The figure shows 2 by 1 multiplexers with 4 input lines A0 to A3 and 4 output lines H0 to H3. The upper (left) multiplexer can take its inputs from serial in (IR) or A0. The last multiplexer (bottom or right) can take its inputs from A3 or serial input (IL). the single line select will select for shift right or left operations.

Arithmetic Logic Shift Unit

Instead of having individual registers performing micro-operations directly, computer systems employ a number of storage registers connected to a unit called Arithmetic Logic Unit (ALU). This unit has 2 operands input ports and one output port and a number of select lines to help in selecting different operations.

The ALU is made of combinational circuit so that the entire register transfer operation from the sources to the destination is performed in one clock cycle.

The arithmetic, logic, and shift circuits known previously will be combined in one ALU with common selection inputs. One stage (bit) of ALSU with its table is shown in next two figures.

As shown the arithmetic and logic units will select their operations simultaneously when S0 and S1 are applied; while S2 and S3 will select one of those unit outputs or a shift left bit stage or shift right bit stage.

Note that one stage arithmetic circuit used here is implemented from figure 1 and the one stage logic circuit is implemented from figure 2.
The circuit shown provides 8 arithmetic operations, 4 logic operations, and 2 shift operations.