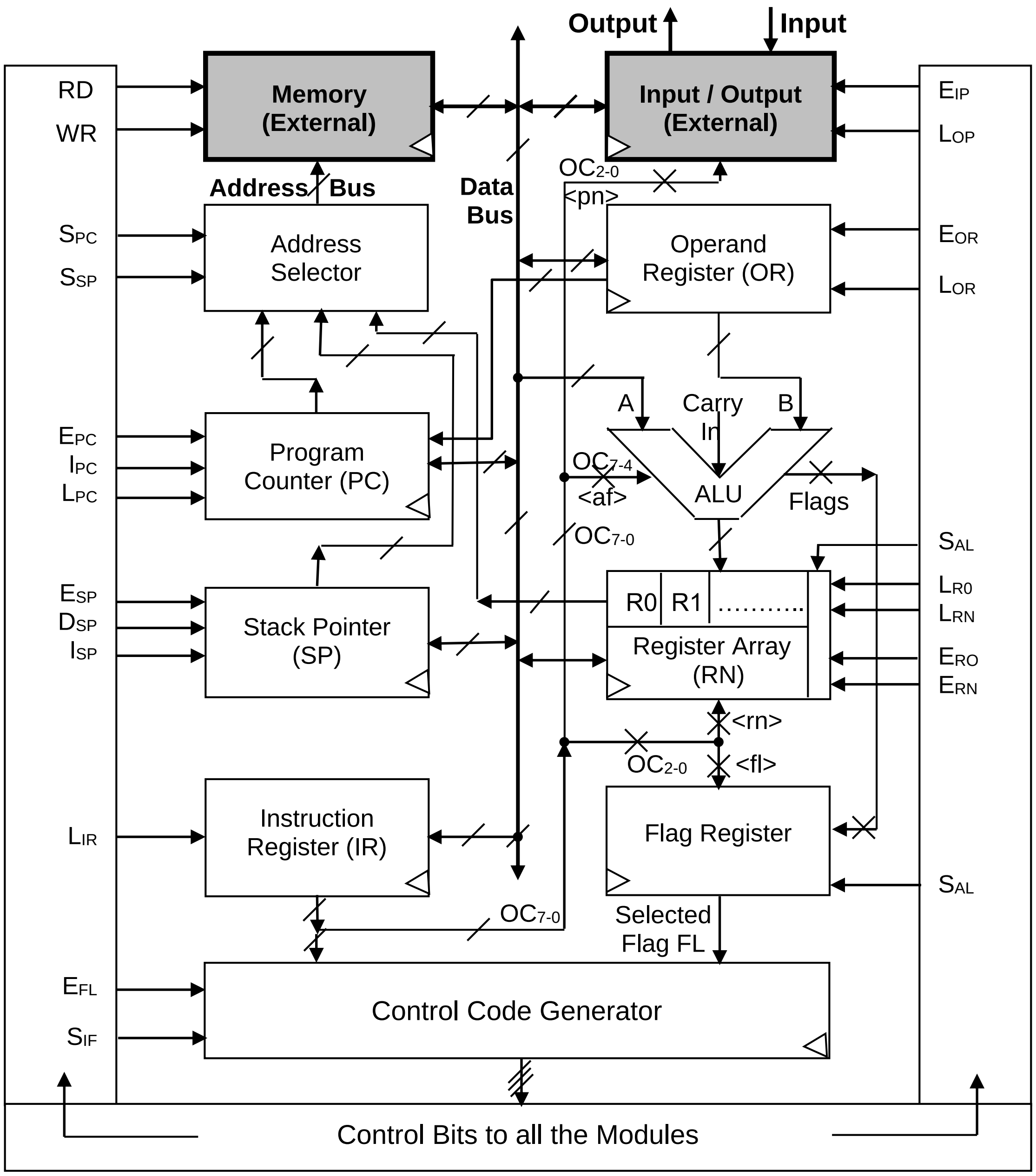
Single Bus Processor Architecture – Final



Nomenclature:

E_{XY} / RD : Enables the Output of the module XY/Memory on to the ***Data Bus***;

L_{XY} / WR : Loads the Data Input applied to module XY/Memory ***at the next Active Clock edge***;

I_{XY} / **D_{XY}** : Increments/Decrements the value stored in module XY;

S_{PC} / S_{SP} :Selects the PC/SP output as the Memory Address (both = 0 selects R0);

S_{IF} : Selects the Instruction Fetch operation, subject to the values of **E_{FL}** and **FL.**

S_{AL} : Selects the ALU output as the Data Input to the Register Array;

The ALU has **16** functions (4-bit Select provided by the leading 4 bits of the Op Code):

Unary **8** [Codes 0 – 7]: Zero, A, A’, B, A + 1, A – 1, Shift Left & C ← MSB, Shift Right & C ← LSB.

Arithmetic **4** [Codes 8 – B]: A + B, A – B, A + B + C, A – B – C [Codes 8 – B].

Logic **4** [Codes C – F]: A AND B, A OR B, A XOR B, A’ XOR B.

The ALU generates **4** flags – Zero (Z), Carry (C), Sign and Parity. Flags are not affected by the Unary Logic functions. Only the **C** flag is affected by the Left/Right Shift function. **All** flags are affected by all other ALU functions. The selected Flag **FL** = **Z / NZ / C / NC / P** (Positive) **/ M** (Negative) **/ PO** (Parity Odd) **/ PE** (Parity Even), depending on the value of **<fl>** (**0 / 1 / 2 / 3 / 4 / 5 / 6 / 7**).

The ***FETCH*** cycle is initiated ***at the next Active Clock edge*** **if (FL’ + E_{FL}’) •S_{IF} = 1.**

Basic Processor Modules

The simple Processor Architecture developed in the class is based on a single internal Data Bus, which is just a set of connecting wires used to carry multi-bit data, the number of data bits being a power of 2 (8/16/32/64). All data transfers from one module to another within the processor take place via the Data Bus. Only one module is allowed (***Enabled***) to put data on the Data Bus at any given time. This is achieved by making the outputs of all modules ***Tri-State***; each line of the Data Bus is either at a level representing logical ‘1’, or at a level representing logical ‘0’, or not connected to any data source and hence remaining ***“floating”***. To transfer data from module ***XY*** to module ***PQ***, the required data path is set up by activating the ***Enable*** control of module ***XY*** (E_{XY} = 1) and the ***Load*** control of module ***PQ*** (L_{PQ} = 1). Data transfer takes place when the ***Clock*** pulse, common to all modules of the Processor, is applied. The exact instant of time when the data actually gets loaded into module ***PQ*** is marked by one of the two transitions (LOW$\rightarrow$HIGH or HIGH$\rightarrow$LOW) of the Clock pulse. The polarity of this Clock transition is chosen by the designer, and is referred to as the ***Active Edge*** of the Clock pulse.

The same Data Bus is extended outside the Processor to provide the path for data transactions between the Processor and the external world. Two important external modules with which the Processor has to interact are the **Input / Output Port** and **Memory**, the basic interaction for both being either to take data from or to send data to these modules. For the sake of simplicity, we will consider only one **Input Port** and one **Output Port**. The **Memory**, on the other hand, consists of a number of locations where information can be stored, and in order to access any Memory location, the Processor has to provide the ***Address*** of the location. Thus the Processor has to have modules for handling the data as well as modules for handling the Memory address. Though the Memory may be one single physical unit, it consists of three logically distinct functional units for storing three types of information pertinent to the operation of the Processor:

- **Program Memory** – stores the user’s program before running it, the program consisting of an ordered sequence of ***Instructions***,
- **Data Memory** – stores data values as required or generated by the user’s program in the course of the execution of the program,
- **Stack Memory** – stores program memory addresses where the program has to ***return*** after going on a ***branch*** to a different segment of the user’s program, in order to ensure correct program flow.

As we shall see, the Memory addresses for these three functional units are generated by three distinct modules.

Each ***Instruction*** reflects a simple task that is functionally tangible and is within the capability of the Processor to perform in a few steps (Clock pulses). An ***Instruction*** can consist just of an ***Op***(eration) ***Code***, which is a binary code denoting the task performed by an Instruction, or of an ***Op Code*** followed by one or more ***Operands***, which provide numerical data needed for the execution of the Instruction. The Op Code and each of the Operands is stored in one location of the Memory For the sake of simplicity, we will restrict the number of Operands to one only, and hence each Instruction will occupy one or two Memory locations.

Let us now list and briefly discuss the functions of the different modules that constitute our simple Processor.

A. <u>Data Handling Modules</u>

- **Programmable ALU**
  The arithmetic and logic functions of the ALU are selectable by a multi-bit select input ***<af>*** provided by the Op Code. The ALU generates four 1-bit ***Flags*** (***Z***, ***C***, ***S***, ***P***) indicating the output conditions Zero output, output with Carry, Sign (most significant bit of the output) and Parity bit of the output.

- **Operand Register (OR)**
  For binary operations (using two inputs) of the ALU, one of the ALU inputs has to be loaded into **OR** before ALU operation. This Register is used by the Processor internally, and cannot be directly accessed by the user's program.

- **Register Array (RN)**
  This is a set of general-purpose Registers accessible through the Data Bus for fast data storage and retrieval. The ALU output can be loaded into any of these Registers. The Register Select input ***<rn>*** is provided by the Op Code.

B. <u>Address Handling Modules</u>

- **Program Counter (PC)**
  This Register provides the **Program Memory** Address. Its content is incremented after every fetch of Op Code / Operand from the Program Memory. Branch Instructions load **PC** with the Branch Address. **PC** thus keeps track of the program flow.

- **Stack Pointer (SP)**
  This is an up-down counter which points to the top of the **Stack**. Its content is incremented /decremented after every Push/Call or Pop/Return Instruction.

- **Accumulator (R0)**
  The **Accumulator R0** is a special Register that forms part of the Register Array, and is used to provide the **Data Memory** Address.

- **Address Selector**
  This module selects the address source (**PC/SP/R0**) for the Memory, depending on the Memory segment (**Program/Stack/Data**) being accessed.

C. <u>Control Modules</u>

- **Instruction Register (IR)**
  The Op Code fetched from the Program Memory is loaded here, and its output is fed to the **Microprogram Sequencer** for generating the Control Signals. The ALU Function code ***<af>***, the Register Select code ***<rn>*** and the Flag Select code ***<fl>*** are either directly available as parts of the Op Code or obtained from the Op Code by decoding it suitably.

- **Flag Register**

  The four **Flags** generated by the ALU are stored in the **Flag Register**. One of these four bits and their complements can be selected by the Flag select input ***<fl>*** provided by the Op Code and given as the Condition Code.

- **Control Code Generator**
  This block generates the sequence of **Control** bits necessary for making each of the functional blocks described above function as required by the Instruction represented by the Op Code held in **IR**, also taking into account the Condition Code, if the Op Code so requires (E_{FL} = 1).

Example of Program Flow with Branching and Return

Program Segment 1		Program Segment 2		Program Segment 3	
Memory Location	**Instruction Type**	**Memory Location**	**Instruction**	**Memory Location**	**Instruction**
40	Data Transfer	50	Data Transfer	60	ALU operation
41	ALU operation	51	ALU operation	61	Data Transfer
42	ALU operation	52	Branch with Return	62	ALU operation
43	Branch with Return	53	Return	63	Data Transfer
44	Data Transfer			64	Return
45	......				

Let the initial value of [***SP***] (value stored in the Stack Pointer) = ***FF***. Then the sequence of values that [***PC***] and [***SP***] will have as the program runs would be as follows, with the address values stored in the Stack as indicated.

[PC]	[SP]	[FF]	[FE]	[FD]
40	FF	Pre-existing value	Pre-existing value	Pre-existing value
41-43	FF	Pre-existing value	Pre-existing value	Pre-existing value
50	FE	Pre-existing value	44	Pre-existing value
51-52	FE	Pre-existing value	44	Pre-existing value
60	FD	Pre-existing value	44	53
61-64	FD	Pre-existing value	44	53
53	FE	Pre-existing value	44	53
44	FF	Pre-existing value	44	53
45	FF	Pre-existing value	44	53

Sl.	Instruction	Type	Action	Op Code
-	Fetch OC	Every instruction	[IR] ← [[PC]]	-
1	NOP	-	No action	0000 0000
2	CLR	ALU Operation	[<rn>](n = 0, 1, …7) ← 0 (all registers cleared)	0000 0001
3	CLC	ALU Operation	C ← 0	0000 0010
4	JUD <od>	Uncond. Branch	[PC] ← <od>	0000 0011
5	JUA	Uncond. Branch	[PC] ← [R0]	0000 0100
6	CUD <od>	Uncond. Branch	[[SP]]← [PC], [SP] ← [SP]–1, [PC] ← <od>	0000 0101
7	CUA	Uncond. Branch	[[SP]]← [PC], [SP] ← [SP]–1, [PC] ← [R0]	0000 0110
8	RTU	Uncond. Branch	[SP]←[SP]+1, [PC]←[[SP]+1]	0000 0111
9	JCD <fl> <od>	Conditional Branch	If <fl> = 1: [PC] ← <od>	0000 1<fl>
10	LSP	Data Transfer	[SP] ← [R0]	0001 0000
11	MVD <rn>*	Data Transfer	[<rn>] ← [R0] (n ≠ 0)	0001 0<rn>
12	RSP	Data Transfer	[R0] ← [SP]	0001 1000
13	MVS <rn>*	Data Transfer	[R0] ← [<rn>] (n ≠ 0)	0001 1<rn>
14	NOT <rn>	ALU Operation	[<rn>] ← [<rn>]’	0010 0<rn>
15	JCA <fl>	Conditional Branch	If <fl> = 1: [PC] ← [R0]	0010 1<rn>
16	CCD <fl> <od>	Conditional Branch	If <fl> = 1: [[SP]]← [PC], [SP] ← [SP]–1, [PC] ← <od>	0011 0<fl>
17	CCA <fl>	Conditional Branch	If <fl> = 1: [[SP]]← [PC], [SP] ← [SP]–1, [PC] ← [R0]	0011 1<fl>
18	INC <rn>	ALU Operation	[<rn>] ← [<rn>] + 1	0100 0<rn>
19	RTC <fl>	Conditional Branch	If <fl> = 1: [SP]←[SP] + 1, [PC]←[[SP]+1]	0100 1<fl>
20	DCR <rn>	ALU Operation	[<rn>] ← [<rn>] – 1	0101 0<rn>
21	MVI <rn> <od>	Data Transfer	[<rn>] ← <od>	0101 1<rn>
22	RLA	ALU Operation	Rotate [R0] Left, C ← MSB, LSB ← C	0110 0000
23	STA <rn>*	Data Transfer	[[R0]] ← [<rn>] (n ≠ 0)	0110 0<rn>*
24	PSH <rn>	Data Transfer	[[SP]] ← [<rn>], [SP] ← [SP]–1	0110 1<rn>
25	RRA	ALU Operation	Rotate [R0] Right, MSB ← C, C ← LSB	0111 0000
26	LDA <rn>*	Data Transfer	[<rn>] ← [[R0]] (n ≠ 0)	0111 0<rn>*
27	POP <rn>	Data Transfer	[SP] ← [SP] + 1, [<rn>] ← [[SP]+1]	0111 1<rn>
28	ADA <rn>	ALU Operation	[R0] ← [R0] + [<rn>]	1000 0<rn>
29	ADI <rn> <od>	ALU Operation	[<rn>] ← [<rn>] + <od>	1000 1<rn>
30	SBA <rn>	ALU Operation	[R0] ← [R0] – [<rn>]	1001 0<rn>
31	SBI <rn> <od>	ALU Operation	[<rn>] ← [<rn>] – <od>	1001 1<rn>
32	ACA <rn>	ALU Operation	[R0] ← [R0] + [<rn>] + C	1010 0<rn>
33	ACI <rn> <od>	ALU Operation	[<rn>] ← [<rn>] + <od> + C	1010 1<rn>
34	SCA <rn>	ALU Operation	[R0] ← [R0] – [<rn>] – C	1011 0<rn>
35	SCI <rn> <od>	ALU Operation	[<rn>] ← [<rn>] – <od> – C	1011 1<rn>
36	ANA <rn>	ALU Operation	[R0] ← [R0] ∧ [<rn>]	1100 0<rn>
37	ANI <rn> <od>	ALU Operation	[<rn>] ← [<rn>] ∧ <od>	1100 1<rn>
38	ORA <rn>	ALU Operation	[R0] ← [R0] ∨ [<rn>]	1101 0<rn>
39	ORI <rn> <od>	ALU Operation	[<rn>] ← [<rn>] ∨ <od>	1101 1<rn>
40	XRA <rn>	ALU Operation	[R0] ← [R0] ⊕ [<rn>]	1110 0<rn>
41	XRI <rn> <od>	ALU Operation	[<rn>] ← [<rn>] ⊕ <od>	1110 1<rn>
42	INA <pn>	Data Input	[R0] ← [<pn>]	1111 0<pn>
43	OUT <pn>	Data Output	[<pn>] ← [R0]	1111 1<pn>

EED206 **Digital Electronics** **Monsoon 2019**

Experiment 8 **Microprocessor Basics**

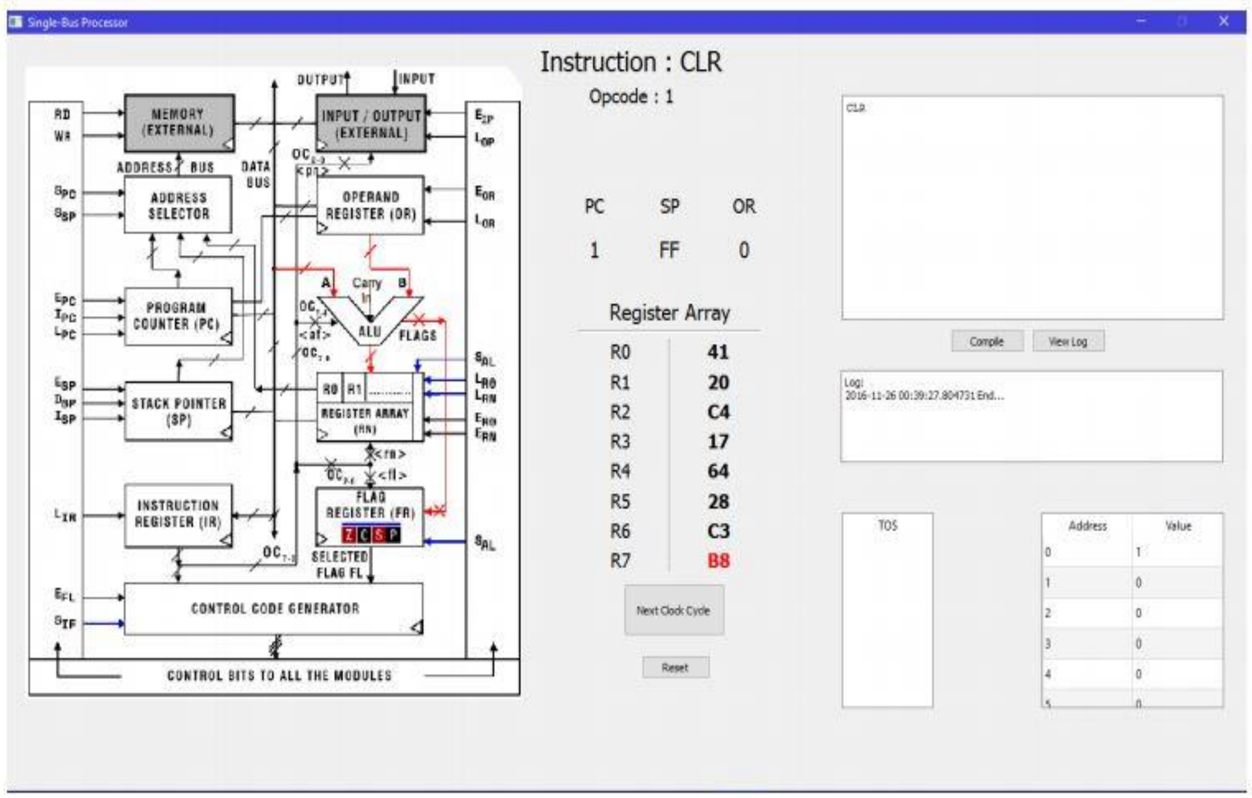
In this experiment, the basic working of a simple Single-Bus Processor will be studied by simulating the design on a Laptop, using an emulation package developed by SNU students of the class of 2014.

A. Installation of the Emulation Package

Download the package to your PC from Blackboard.

1. Download the RNBIP package (.rar file) having the name **RNBIPv1_2_DOS.rar.**
2. Extract the .rar file to create a folder "**RNBIPv1_2_DOS**".
3. Go through "**Pre-Installation Instructions.pdf**" available in the folder.
4. Access the folder "**Python Installation**" from the folder and install the Python software version "**python-3.6.3-amd64**".
5. While installing, select "**Add Python 3.6 to PATH**" checkbox.
   Note: If you have already installed Python software, please make sure that there is no conflict between two versions of Python.
6. Install "**RNBIP.exe**" available in "**RNBIP Installation**" folder.

   Note: During installation of RNBIP, please change installation directory from default OS (C:/) directory to another (D:/ or F:/) directory. Python installation drive should not be similar to that of RNBIP.
7. After proper installation you should see the following screen:



Instruction : CLR

Opcode : 1

PC	SP	OR
1	FF	0

Register Array

R0	41
R1	20
R2	C4
R3	17
R4	64
R5	28
R6	C3
R7	B8

```
Log:
2016-11-26 00:39:27.804731 End...
```

Address	Value
0	1
1	0
2	0
3	0
4	0
5	0

B. <u>Running Simple Programs</u>

1. You can refer to the **Readme** file whenever necessary to understand how the Data Path, the Active Control Bits and the contents of the Registers and Memory are displayed on the screen for every clock pulse.

2. Perform the emulation of the following assembly language Programs one by one, as explained below.

Program 1	
Address	**Instruction**
00	CLR
01	MVI R7 70
03	MVI R1 A1
05	MVS R1
06	STA R7
07	INC R0
09	ADI R7 70
0A	JCD C 00
0C	JUD 06

Program 2	
Address	**Instruction**
00	CLR
01	MVI R4 50
03	ADI R4 FF
05	CCD C 0A
07	NOP
08	JUD 00
0A	DCR R3
0B	RTU

3. Load the program in the Code area and verify that the contents of the memory locations indicated above are the Op Codes corresponding the Mnemonics followed by Operands, as applicable.

4. Run the instructions step by step, each step representing a clock cycle, and at each step, identify from the highlighted Data Path and Control Bits the action taken and hence the type (Fetch / Read / Execute) of the cycle, as explained below:

   i) **Fetch cycle –** Any instruction starts with an Op Code Fetch cycle, where the Memory is addressed by [PC], the contents of the addressed location is placed on the Data Bus, and the Instruction Register is **ready to get loaded** from the Data Bus. Note that the action **[IR]← [[PC]]$_{\text{mem}}$** actually takes place at the **Clock Edge at the end** of the Fetch cycle, and so Emulation shows the **previous Op Code** while showing the Active Control Bits and the Data Path for the Fetch operation.

   ii) **Read cycle –** This is a cycle where any data to be read out, either from the Memory or from any Register, is placed in a Temporary Register before it is used for generating the final result. In our context, the Read cycle, if necessary, will only involve data transfer to the **Operand Register**.

   iii) **Execute cycle –** This is the final cycle of any Instruction, and involves the loading of the final result, created either by the ALU or just by routing data through a chosen Data Path, into the final destination, which can be any **Register** or **Memory**. The Execute cycle may include a Read operation and thus an Instruction may consist only of the Fetch cycle followed by Execute cycle.

5. Fill up the table shown below by writing down, as applicable against each instruction of the program, the actions taken in the Read and the Execute cycles, writing down in parentheses the Control Bits responsible for the action taken in each step. Hence determine the total number of Clock cycles required for each of the instructions.

Instruction	Read cycle	Execute cycle	Clock Cycles

6. Based on your observations for each of the instructions, explain the overall action of the program. How would the execution of Program 2 be affected if `FF` is replaced by `10` in the third line of the Program. [*Hint: Find out how the condition tested in the instruction* `CCD C A` *gets modified.]*

Control Bits

Every data transaction in a Processor requires a Data Path to be established by the activation of selected Control Bits. In the nomenclature adopted by us for the Single-bus Processor, the Control Bits can be of five different types:

(a) ***Enable*** – Denoted generically by E_{XY}, where ***XY*** refers to the Module in question, this category of Control Bits is used to enable Module ***XY*** to place data on the Data Bus, with only one exception – E_{FL}, which enables the selected Flag (***FL***) to decide the next Instruction Fetch. ***RD*** (≡ E_{ML}) is a special Enable bit used to place on the Data Bus (≡ **Read**) the contents of a Memory Location selected by the Address provided by the Address Selector.

(b) ***Load*** – Denoted generically by L_{XY}, where ***XY*** refers to the Module in question, this category of Control Bits is used to connect the Data Bus to the Data Input of module ***XY***, so that the data applied to the Data Bus by the selected Data Source (Module enabled by an appropriate ***Enable*** bit) gets loaded into Module ***XY*** at the next **Active Edge** of the Clock. ***WR*** (≡ L_{ML}) is a special Load bit used to load the data available on the Data Bus (≡ **Write**) into the Memory Location selected by the Address provided by the Address Selector.

(c) ***Select*** – Denoted generically by S_{XY}, where ***XY*** refers to the Module in question, this category of Control Bits is used to select one out of the multiple data sources. Two selections are built into the Op Code – its least significant 3 bits OC_{2-0} (≡ ***<fl>/<rn>/<pn>***) being used the select the **Flag/Register/Port** to be used in the Instruction, and its most significant 4 bits OC_{7-4} (≡ ***<af>***) being used to select the **ALU Function**. S_{IF} is a special Select bit used to initiate the next **Instruction Fetch**, either unconditionally if the Control Bit E_{FL} = 0, or subject to the value of the Flag selected by ***<fl>*** if E_{FL} = 1.

(d) ***Increment*** – Applicable only to ***PC*** and ***SP***, this Control Bit is used to decide whether the contents of ***PC/SP*** will be incremented at the next **Active Edge** of the Clock or not.

(e) ***Decrement*** – Applicable only to ***SP***, this Control Bit is used to decide whether the contents of ***SP*** will be decremented at the next **Active Edge** of the Clock or not.

The actions of the last three types of control bits are straightforward, and the module-wise control bits tabulated on the next page should be sufficient to clarify them. There is scope for some ambiguity in the actions of the **Enable** and **Load** bits on modules that have multiple data paths for Output and Input respectively. Let us therefore examine the data paths associated with such modules one by one in order to understand the actions of the control bits pertaining to these modules.

(a) **Memory** – This module has two data paths, one for the **Address Bus** and the other for the **Data Bus** of the Memory module. The **Address Bus** is permanently connected to the output of the **Address Selector** module, so as to keep the desired Memory Location always selected. What the control bits ***RD*** and ***WR*** do is to connect the **Data Bus** of the Processor either to the output or to the input of the Memory Location so as to set up the data path for the desired operation.

(b) **Program Counter (*PC*)** – Control bits E_{PC} and L_{PC} control data transactions between ***PC*** and any other module via the **Data Bus**. The data path connecting the ***PC*** to the **Address Selector** is, on the other hand, independent of control bits, and so [***PC***] is ALWAYS available as the source for (Program) Memory Address. Thus during the Fetch cycle of every instruction, Memory Address = [***PC***] during the transfer of the Op Code from (Program) **Memory** to the **Instruction Register** and [***PC***] ← [***PC***] + 1 at the end of that Clock cycle. The E_{PC} bit is NOT applicable to this data transaction; NOR is it applicable to the transfer of the Operand from (Program) **Memory** to the **Operand Register** during the execution of an Instruction containing an Operand.

(c) **Stack Pointer (*SP*)** – Like ***PC***, ***SP*** also provides an ALWAYS available source for (Stack) Memory Address to the **Address Selector**. The value provided through this data path is [***SP***] – 1 if D_{SP} = 1, and [***SP***] otherwise; the next Active Clock Edge decrements [***SP***] if D_{SP} = 1, and increments [***SP***] if I_{SP} = 1. Thus for the instructions CUD, CCD and PSH, Memory Address = [***SP***] – 1 during the saving of [***PC***] to (Stack) **Memory** and [***SP***] ← [***SP***] – 1 at the end of that Clock cycle; while for the instructions RTU, RTC and POP, Memory Address = [***SP***] during the transfer of the Return Address from (Stack) **Memory** to ***PC*** and [***SP***] $\leftarrow$ [***SP***] + 1 at the end of that Clock cycle. E_{SP} and L_{SP} control transactions between ***SP*** and any other module via the **Data Bus**. They are NOT applicable to the transactions carried out in the Branch instructions mentioned above.

(d) **Operand Register** (***OR***) – The value (operand) contained in the Operand Register is ALWAYS available as the **B Input** to the ALU, The control bits E_{OR} and L_{OR} are applicable only to data transactions between ***OR*** and any other module via the **Data Bus**.

(e) **Register Array** – The Registers in this module can receive input either from the ALU Output or from the Data Bus, the selection being done by the control bit S_{AL} (S_{AL} = 1 selects ALU output). The output of the Accumulator (***R0***) is ALWAYS available as the (Data) memory Address input to the **Address Selector**. E_{R0} is applicable only for placing [***R0***] on the **Data Bus**.

Module-wise Control Bit Action: ⎍ implies action at the next Active Edge of the Clock

Module	Control Bits	Action of Control Bits when Active (= 1)
Memory	***RD***	Contents of selected Memory location read out on Data Bus
	WR	Data on Data Bus written into the selected Memory Location ⎍
Address Selector	$S_{PC}\ S_{SP}$ = 0 0	[***R0***] selected to provide the address of the Memory Location
	$S_{PC}\ S_{SP}$ = 0 1	[***SP***] selected to provide the address of the Memory Location
	$S_{PC}\ S_{SP}$ = 1 0	[***PC***] selected to provide the address of the Memory Location
Program Counter	E_{PC}	Contents of the Program Counter [***PC***] read out on Data Bus
	L_{PC}	Data on Data Bus written into the Program Counter ⎍
	I_{PC}	Contents of the Program Counter [***PC***] incremented ⎍
Stack Pointer	E_{SP}	Contents of the Stack Pointer [***SP***] read out on Data Bus
	L_{SP}	Data on Data Bus written into the Stack Pointer ⎍
	I_{SP}	Contents of the Stack Pointer [***SP***] incremented ⎍
	D_{SP}	Contents of the Stack Pointer [***SP***] decremented ⎍
Instruction Register	L_{IR}	Data on Data Bus (Op Code) written into the Instruction Register ⎍
Input Port	E_{IP}	Contents of the selected Input Port [***PN***] read out on Data Bus
Output Port	L_{OP}	Data on Data Bus written into the selected Output Port ⎍
Operand Register	E_{OR}	Contents of Operand Register [***OR***] read out on Data Bus
	L_{OR}	Data on Data Bus written into the Operand Register ⎍
Register Array	E_{R0}	Contents of Accumulator [***R0***] read out on Data Bus
	E_{RN}	Contents of selected Register [***RN***] read out on Data Bus
	$L_{R0}\ L_{RN}$ = 0 1	Input Data written into the selected Register ***RN*** ⎍
	$L_{R0}\ L_{RN}$ = 1 0	Input Data written into the Accumulator ***R0*** ⎍
	$L_{R0}\ L_{RN}$ = 1 1	Input Data written in parallel into ALL the Registers ***R0, R1, R2...R7.*** ⎍
	S_{AL} = 0	Data on Data Bus selected as the Input Data for the Register Array
	S_{AL} = 1	ALU Output selected as the Input Data for the Register Array
Control Code Generator	E_{FL}	Conditional Branch Instruction
	S_{IF}	Next Instruction Fetch initiated either if E_{FL}= 0 or if ***FL*** = 0