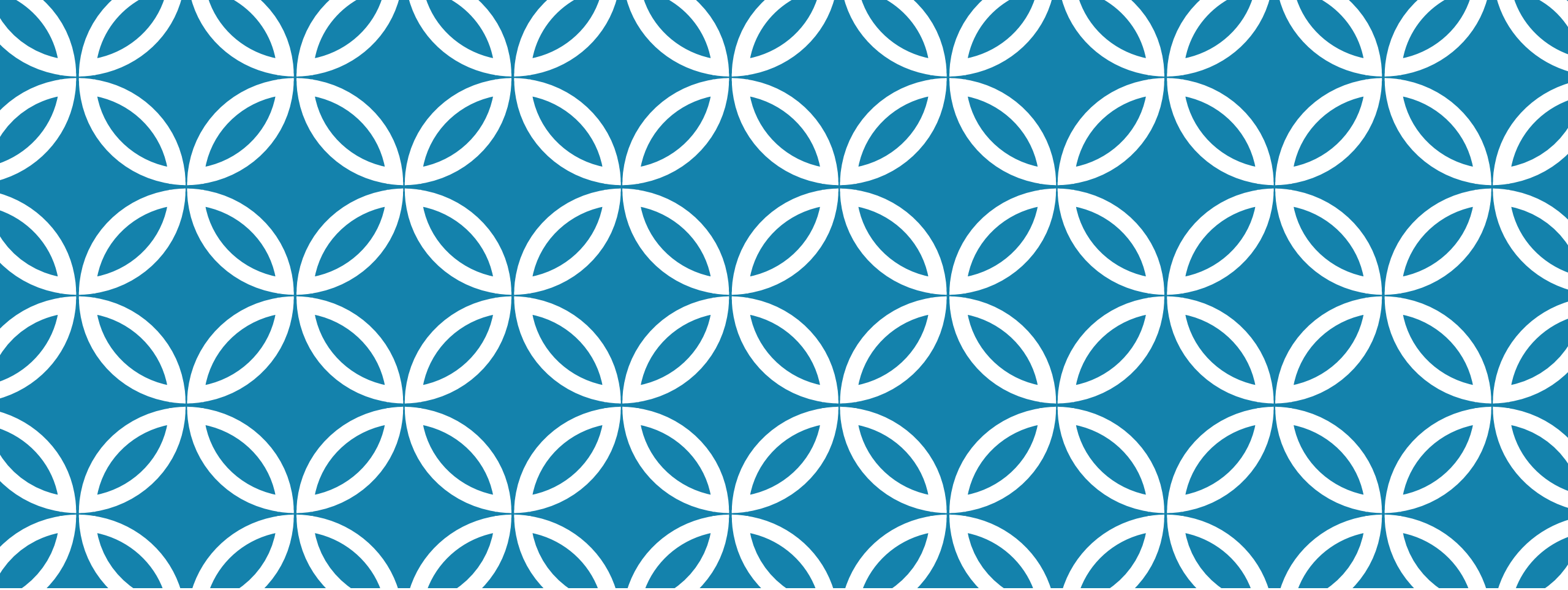


THE MICROARCHITECTURE

I4GIC

FUNCTIONS AND CHARACTERISTICS OF INSTRUCTIONS

It was already mentioned that the instructions directly executable by any processor are very primitive in nature. In general, they are as simple as: 'copy a byte from here to there' or 'add contents of these two registers', or something similar to these.

In general instructions can be placed in 3 categories:

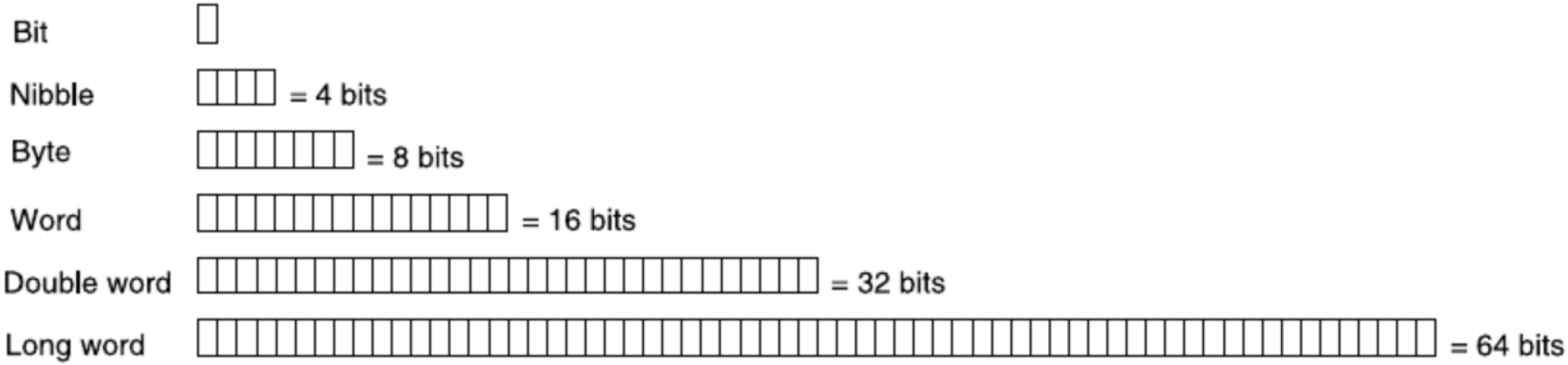
1. Data move type instructions
2. Arithmetic and logical instructions
3. Program ow control and machine control instructions.

DATA MOVE TYPE

These instructions are most frequently used to copy data from one place to another. The source and destination of movements may be registers within the processor or some external memory locations or a combination of both.

Immediate data, presented as a part of the instruction, may also be loaded to some destination defined by these instructions.

Data is passed through the bus and note that internal CPU bus is much faster than external bus. That's why data movement between registers-registers or between CPU's components are always recommended.



To be simple, CPU prefer handling data with the same size (8 bits or 16 bits or …) but in real practice it can handle different length of data. 8 bits CPU can normally handle 8 bits data, 4 bits data, 2 bits data and 1 bit data.

It is getting more complicated the more data length a CPU can handle but the possibility is there.

ARITHMETIC AND LOGICAL TYPE

Most processors offer instructions for four basic arithmetic operations (add, subtract, multiply and divide) with signed and unsigned integers and essential logical operations, e.g., AND, OR, XOR, NOT, with shift and rotate instructions.

These instructions normally required some status flags such as carry, signed, zero, …

PROGRAM FLOW CONTROL TYPE

Program flow control instructions form the third major group of processor instructions.

Generally, they are sub-divided as conditional and unconditional branching and subroutine call and return.

Branching instructions act depend on the status registers (flags)

Carry and zero are most widely used branching conditions. Instructions related with branching as per the condition of carry flag or zero condition are offered by all processors

Instruction	Function
Copy	Copy data from register to register
Load	Load data from memory to register. Load immediate data in register
Store	Save data from register to memory
Clear	Load register by 0s
Set	Load register by 1s
Exchange	Exchange data between two registers or between a register and a memory location
Push	Save data on stack-top indicated by stack pointer
Pop	Load data from stack-top into register
Input	Read data from input port and store within indicated register
Output	Write data from indicated register into output port

Common data transfer type instructions for processors

Instruction	Function
Add	Add two operands
Add with carry	Add two operands and content of carry ag
Subtract	Subtract one operand from another
Subtract with carry	Subtract one operand from another along with carry
Multiply unsigned	Unsigned multiplication of two operands
Multiply signed	Signed multiplication of two operands
Divide unsigned	Unsigned division of two operands
Divide signed	Signed division of two operands
Increment by one	Increment an operand by one. Generally used for address manipulations.
Decrement by one	Decrement an operand bby one. Generally used for address manipulations.

Common arithmetic type instructions for processors

Instruction	Function
AND	Logically AND two operands
OR	Logically OR two operands
XOR	Logically XOR two operands
NOT	Complement the operand
Shift right	Shift operand right one bit. LSB is lost. New constant introduced at MSB.
Shift left	Shift operand left one bit. MSB is lost. New constant introduced at LSB.
Rotate right	Shift operand right one bit. LSB is shifted to MSB.
Rotate left	Shift operand left one bit. MSB is shifted to LSB.
Compare	Compare two operands and re ect the result through ags
Test	Test ag bit(s)

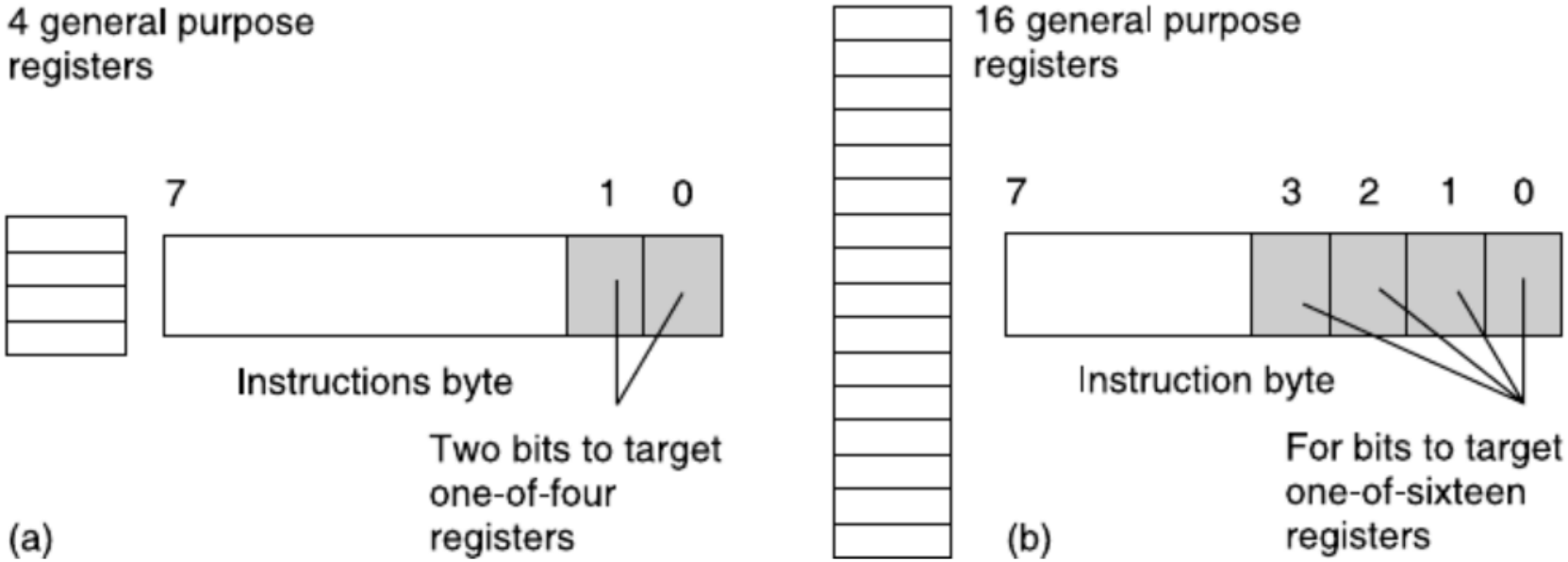
Common logical type instructions for processor

Instruction	Function
Branch unconditional	Jump to the indicated address
Branch conditional	Test condition and if condition is true then jump to indicated address
Call a subroutine	Save PC on stack-top and branch to indicated address
Return from subroutine	Reload PC by address saved on stack-top
Return from interrupt	Enable interrupts and reload PC from stack-top
No operation	Do nothing
Wait	Wait for a signal input
Halt	Stop functioning of the processor
Skip next instruction	Execute the instruction immediately after the next instruction
Branch relative to PC	Add PC with an offset and branch there

Common program flow control type instructions for processors

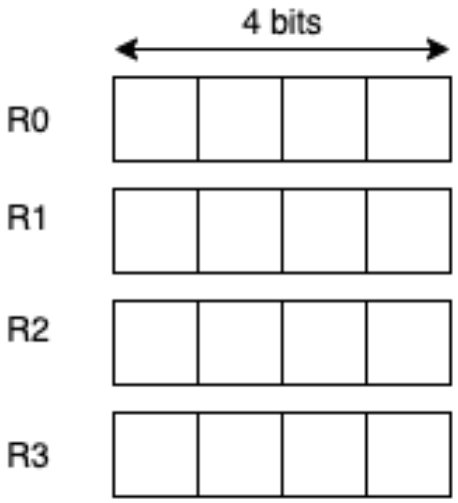
Schematic of a few formats of instructions

In general, an instruction is always transferred or store once together. But it is also possible to do so in different transaction or locations. Anyway, it is going to be complicated so avoid it unless you don't have the choices.



Register field variation for processor with (a) four and (b) sixteen registers

LET'S IMAGINE WE HAVE THE FOLLOWING COMBINATION



Only 4 instructions:

1. Load accumulator by a register
2. Save accumulator within a register
3. Add a register with accumulator, leaving its result in accumulator
4. Subtract a register from accumulator, leaving its result in accumulator

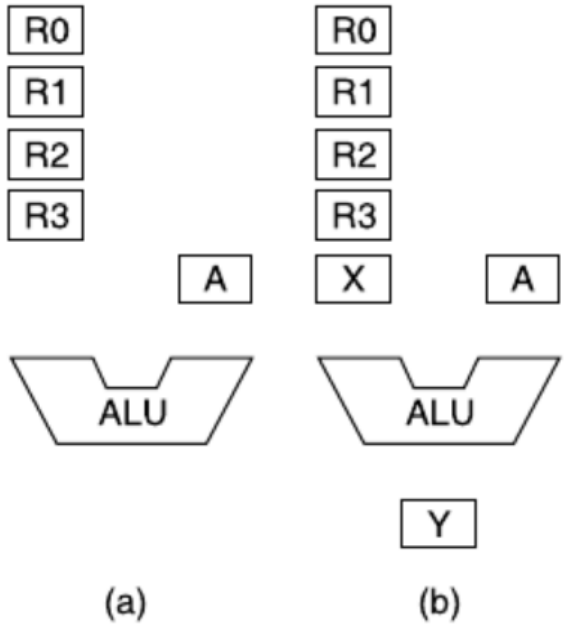
We are not considering the carry-in or carry-out in the arithmetic operations.

1. Set your expectation

3 2 1 0		3 2 1 0		3 2 1 0		3 2 1 0	
0 0 0 0	Load A by R0	0 1 0 0	Save A in R0	1 0 0 0	Add A with R0	1 1 0 0	Subtract R0 from A
0 0 0 1	Load A by R1	0 1 0 1	Save A in R1	1 0 0 1	Add A with R1	1 1 0 1	Subtract R1 from A
0 0 1 0	Load A by R2	0 1 1 0	Save A in R2	1 0 1 0	Add A with R2	1 1 1 0	Subtract R2 from A
0 0 1 1	Load A by R3	0 1 1 1	Save A in R3	1 0 1 1	Add A with R3	1 1 1 1	Subtract R3 from A

2. Visualizing Data-flow and Registers

Our ALU will have two input registers and one output register. We have designated the accumulator (register A) as one of the two input register for the ALU. Therefore, we shall need one more input register and we designate that register as X, which would be a temporary register. To hold the output from the ALU, we include another register in our design, say register Y. Both X as well as Y would be 4-bit registers, like all other registers in our example case

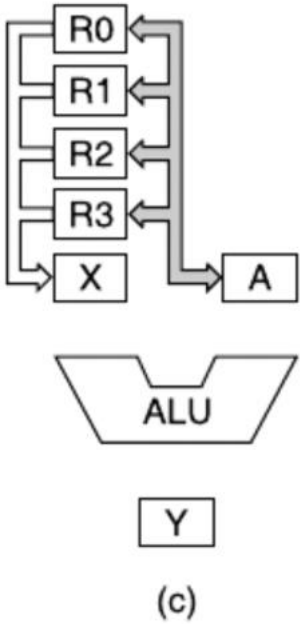


Register organization

3. Data Path Design

The next step is to provide all necessary paths so that the data can ow properly from one register to another as per our requirements.

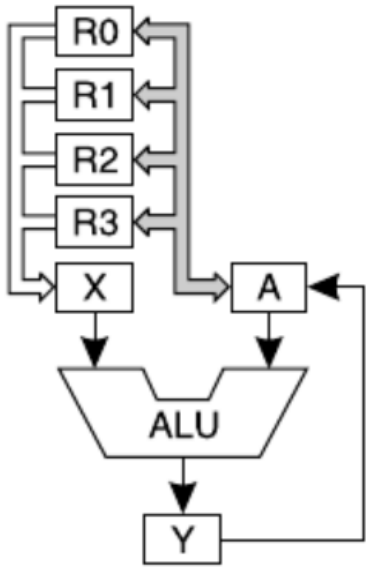
Considering first two instructions, we note that we must provide a bidirectional data path between the accumulator and all four registers, so that accumulator may be loaded from any register or may be stored in any register.



Bidirectional data path of accumulator

For addition and subtraction operations, apart from accumulator, another set of data must be available in the temporary register X, which should be copied from the indicated register as per the instruction. Therefore, there must be another bus connecting the temporary register X with all four registers from R0 to R3.

This data bus would be unidirectional , from register-set (R0 – R3) to X-register as in no case data would be copied from this temporary register (X) to any other register (R0 – R3).



(d)

To start with, note that we have not provided any mechanism of loading any immediate data (using immediate addressing mode) to the accumulator or four other registers (R0 – R3), for the purpose of focusing on essential issues.

Next, we have avoided indicating any flag register, which is considered to be essential for any ALU, for keeping the problem a simple one. Furthermore, the data path, provided by us is not an optimum one, but takes care of all necessary data flow, as per the demand of our initial assumptions regarding the problem.

In total, we have provided six data paths. Only one of these six paths is bidirectional, and the remaining five are unidirectional. Note that the bidirectional data path is also capable of copying the data from any register (R0 – R3) to another register (R0 – R3), which we are not implementing for our present example problem.

From accumulator, there are three different data paths. One is the bidirectional data path for registers R0 – R3. The second one is from the accumulator to the ALU. The third one is to the accumulator from Y register. These last two paths would be used only during the ALU operations.

From each of the four registers (R0 – R3), there are two data paths, one for communicating with the accumulator and another to move the selected register's data to X-register. The second path would be necessary before any ALU operations.

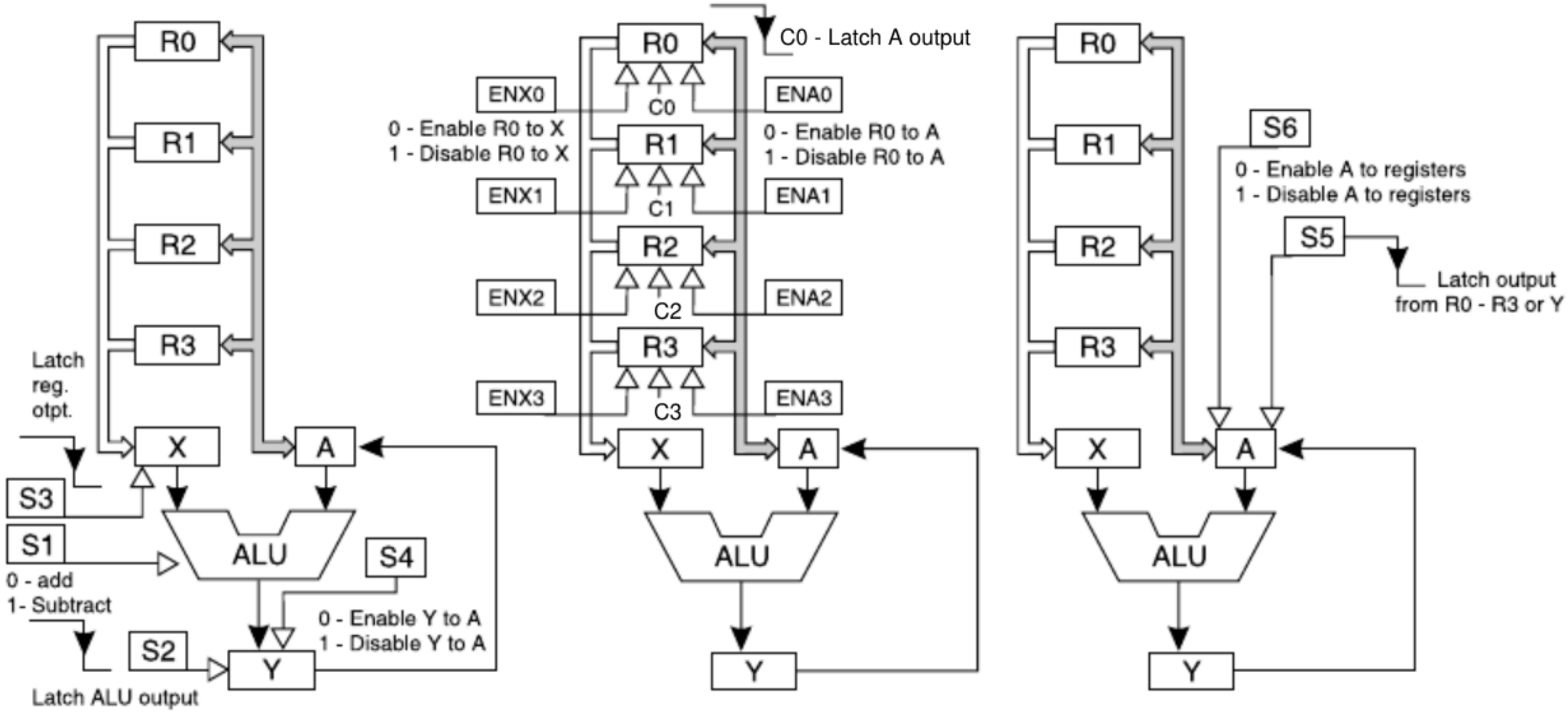
For the X-register, two separate data path exists, one to get the data from any one of the four registers R0 – R3. Another one is to load it within the ALU. For Y-register also there are two separate data paths, one to receive the result from ALU and another to send the data to the accumulator. All these ensure that we have provided adequate data paths to maintain the data ow, as per the requirement of the problem statement.

4. Control signal requirement

To make our unit a working one, we need to design all necessary control signals for data propagation or proper sequential flow of data.

What is a control signal?

- In this case, control signal indicates those signals whose change of logic level would allow or disallow (enable or disable) the data flow from one point (register etc.) to another.


R0
R1
R2
R3
Latch
reg.
otpt.
X
A
S3
S1
ALU
S4
0 - add
1- Subtract
0 - Enable Y to A
1 - Disable Y to A
S2
Y
Latch ALU output
C0 - Latch A output
ENX0
ENA0
C0
0 - Enable R0 to X
1 - Disable R0 to X
0 - Enable R0 to A
1 - Disable R0 to A
ENX1
ENA1
C1
ENX2
ENA2
C2
ENX3
ENA3
C3
S6
0 - Enable A to registers
1 - Disable A to registers
S5
Latch output
from R0 - R3 or Y

Name	Location	Function	Remarks
S1	ALU	ALU function control	0-add, 1-subtract
S2	Y-register	Latch ALU output in Y	Falling edge
S3	X-register	Latch Rn output in X	Falling edge
ENX0–ENX3	R0–R3	Enable Rn for X	0-enable, 1-disable
ENA0–ENA3	R0–R3	Enable Rn for accumulator	0-enable, 1-disable
C0–C3	R0–R3	Latch accumulator output in Rn	Falling edge
S4	Accumulator	Enable accumulator for R0–R3	0-enable, 1-disable
S5	Accumulator	Latch Rn output in accumulator	Falling edge
S6	Accumulator	Latch Y output in accumulator	Falling edge

Summary of control signals

4. Need micro instructions

- What is meant by microinstructions?

To implement any instruction, several steps would be necessary. These steps are designated as micro-steps and the necessary instructions to generate these micro-steps are known as microinstructions.

For example, let us take the first instruction: Load accumulator by a register and assume that the instruction is referring a general purpose register R0. We find that the following micro-steps (and sequence) are necessary to implement the instruction Load accumulator by R0:

1. Make ENA0 as 0 to enable output of R0 to reach accumulator.
2. Latch the input from R0 to accumulator by a high to low signal at S5.

Note that the sequence is also important in this case. If we activate S5 first and then activate ENA0, we shall not be able to receive the correct data within accumulator.

Instruction	Step 1	Step 2	Step 3	Step 4
Load A by R0	ENA0 = 0	S5 1 to 0	S5 0 to 1	ENA0 = 1
Load A by R1	ENA1 = 0	S5 1 to 0	S5 0 to 1	ENA1 = 1
Load A by R2	ENA2 = 0	S5 1 to 0	S5 0 to 1	ENA2 = 1
Load A by R3	ENA3 = 0	S5 1 to 0	S5 0 to 1	ENA3 = 1
Explanation of steps (microinstructions)	Enable target register	Latch data in accumulator	Reset S5	Disable target register

Micro-steps for Loading Accumulator

Instruction	Step 1	Step 2	Step 3	Step 4
Save A in R0	S6 = 0	C0 1 to 0	C0 0 to 1	S6 = 1
Save A in R1	S6 = 0	C1 1 to 0	C1 0 to 1	S6 = 1
Save A in R2	S6 = 0	C2 1 to 0	C2 0 to 1	S6 = 1
Save A in R3	S6 = 0	C3 1 to 0	C3 0 to 1	S6 = 1
Explanation of steps	Enable accumulator	Latch data in target register	Reset latching signal	Disable accumulator

Micro-steps for saving accumulator

Instruction	1	2	3	4	5	6	7	8	9	10
Add A with R0	ENX 0 = 0	S3 1 to 0	S3 0 to 1	S1 = 0	S2 1 to 0	S2 0 to 1	S4 = 0	S5 1 to 0	S5 0 to 1	S4 = 1
Add A with R1	ENX 1 = 0	S3 1 to 0	S3 0 to 1	S1 = 0	S2 1 to 0	S2 0 to 1	S4 = 0	S5 1 to 0	S5 0 to 1	S4 = 1
Add A with R2	ENX 2 = 0	S3 1 to 0	S3 0 to 1	S1 = 0	S2 1 to 0	S2 0 to 1	S4 = 0	S5 1 to 0	S5 0 to 1	S4 = 1
Add A with R3	ENX 3 = 0	S3 1 to 0	S3 0 to 1	S1 = 0	S2 1 to 0	S2 0 to 1	S4 = 0	S5 1 to 0	S5 0 to 1	S4 = 1
Explanation of steps	Enable target register	Latch data in X	Reset S3	For adding	Latch ALU output in Y	Reset S2	Enable Y	Latch result in A	Reset S5	Disable Y

Micro-steps for adding accumulator with a register

Instruction	1	2	3	4	5	6	7	8	9	10
Subtract R0 from A	ENX 0 = 0	S3 1 to 0	S3 0 to 1	S1 = 1	S2 1 to 0	S2 0 to 1	S4 = 0	S5 1 to 0	S5 0 to 1	S4 = 1
Subtract R1from A	ENX 1 = 0	S3 1 to 0	S3 0 to 1	S1 = 1	S2 1 to 0	S2 0 to 1	S4 = 0	S5 1 to 0	S5 0 to 1	S4 = 1
Subtract R2 from A	ENX 2 = 0	S3 1 to 0	S3 0 to 1	S1 = 1	S2 1 to 0	S2 0 to 1	S4 = 0	S5 1 to 0	S5 0 to 1	S4 = 1
Subtract R3 from A	ENX 3 = 0	S3 1 to 0	S3 0 to 1	S1 = 1	S2 1 to 0	S2 0 to 1	S4 = 0	S5 1 to 0	S5 0 to 1	S4 = 1
Explanation of steps	Enable target register	Latch data in X	Reset S3	For sub-tracting	Latch ALU output in Y	Reset S2	Enable Y	Latch result in A	Reset S5	Disable Y

Micro-steps for subtracting a register from accumulator