A **register** is a *sequential digital circuit* capable of storing and updating a multi-bit binary value synchronously with a clock signal.

Key word: **synchronously**.

An *n-bit register* is implemented using **n D flip-flops**, all sharing a **common clock** and **common control logic**.

This common clock is *critical* in CPU design.

SAP-1 employs a **single shared 8-bit bus**, and bus contention is avoided by ensuring that **only one register enables its tri-state output buffer at any given time**, as dictated by the control unit.

Although universal registers support shifting, SAP-1 registers are intentionally simplified to reduce datapath complexity.

Register is a sequential device - be able to store and be able to manipulate a collection of indivdiual bits.
- Registers are a collection of D flipflops.

In order to create a register, we require a multiplexer, and a D flipflop, and based on the select lines, we perform the desired function.

Select Inputs		Function
S1	S0	
0	0	Store
0	1	Shift Left
1	0	Shift Right
1	1	Load

The storage process for a D flipflop, requires the clock signal to be either high or low, depending on whether the fliflop is positive edge triggered or negative edge triggered, but, in our circuit, we require the clock to be synchronized across all individual components, and therefore, we route the value stored at the flipflop back to the input.

For a SAP-1 architecture, only one register drives the 8-bit bus, one at a time.

In a synchronous system, it is preferred that all the digital blocks receive the clock pulse at the same time.

How is the Control Unit an FSM?

The control unit of the SAP-1 CPU is implemented as a finite state machine, where each state corresponds to a micro-operation in the instruction cycle. State transitions are synchronized by the system clock, and each state asserts a unique set of control signals that govern register transfers and ALU operations.

In Digital Logic Design, a control unit is:

A sequential controller that issues control signals in a specific time order to orchestrate datapath operations.

OCTAL BUS TRANSCEIVER - 74LS245
By using this tri state logic, it enables us to have the enable line be low, so that, only one of the registers is enabled, at at time, so that we don't have any conflicts on the 8-bit bus.

11 to 8 – these will be for the Opcode
7 – 0 – these will be the reserved switches for the actual 8 bit data
BTNR (M17 Port) – this is the write strobe
BTND (P18 Port) – this is the clear switch (clears the RAM)
Operation: Firstly, load the data into the two general purpose registers. Then, run the Opcode for whatever instructions needs to be run (e.g. ADD, SUB, etc.). This will perform the operation upon the two operands in the general purpose registers and save it in the RAM. Whatever is saved in the RAM, will be output and shown on the 7-SEGMENT Display.

8-bit Bus

8-bus is basically 8 wires/lines used for data transfer between modules.

Rules:

i. At a single time instance only one module will send data to the bus, and only one module will receive that data.

ii. Two modules can never send data to the bus at the same time (bus contention).

iii. When a module is sending data to the bus, all other modules' outputs must be disconnected (we use tri-state buffers).
We use a tri-state buffer for the output for our modules, in order to avoid bus contention.

Memory Address Register (MAR)

•It’s a 4-bit register.
•It stores the memory address of RAM from which data will be fetched.
•This 4-bit address is obtained via the bus from the Program Counter and then stored.
•This stored address is sent to the RAM where data or instructions are read from.
•Control bits:

•Memory Address In (MI): Stores the current values of the bus into the Memory address register (MAR).

The accumulator and the B-register are connected to the ALU, so that the data can be sent to the ALU.

What a *standard* SAP-1 ALU does (baseline)

In classical SAP-1 (as taught in DLD and Ben Eater):

- The ALU supports:
    - **8-bit addition**
    - **8-bit subtraction (A − B)**

Control signals:

Signal	Meaning
SU	Selects operation (0 = Add, 1 = Subtract)
EO	Enables ALU output onto the system bus

Internally:

- Subtraction is implemented using **two’s complement**:

  A−B=A+(∼B+1)A - B = A + (\sim B + 1)A−B=A+(∼B+1)
- The ALU output is placed on the bus **only when EO is high** (otherwise it must be high-impedance).

---

2. High-level structure of *your* ALU

Your ALU follows the **same mathematical principle**, but the **hardware organization is more modular and explicit** than the minimal SAP-1.

Key idea:

> Instead of conditionally inverting B inside the adder, you **route B through demultiplexers and multiplexers** before the adder.

This makes the dataflow **very clear** and easy to debug in simulation.

The custom ALU implements 8-bit addition and subtraction using explicit demultiplexing and two's complement logic, with tri-state output control to support a shared SAP-1 system bus.

Two's Complement Logic

```
assign comp = ~dmux1 + 8'b0000_0001;
```

Syntax breakdown:

- `assign` → continuous assignment (combinational)
- `~` → bitwise NOT
- `8'b0000_0001` → 8-bit binary constant

Hardware meaning:

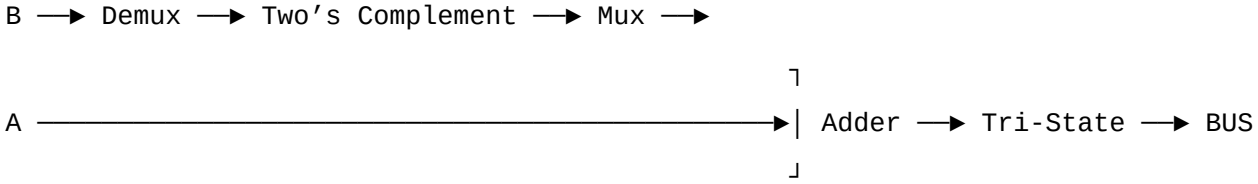
- If subtraction:
  - `dmux1 = B`
  - `comp = ~B + 1`
- If addition:
  - `dmux1 = 0`
  - `comp = 0`

📌 This explicitly implements:

$-B = \sim B + 1$-B=∼B+1

How Everything Is Linked (Signal Flow)

Step-by-step flow:



Control Signals:

- `sub` controls:
    - Demux
    - Mux
    - Subtraction logic
- `out_en` controls:
    - Bus connection

Specification of Instruction Set

•An instruction in the SAP-1 architecture consists of 1 byte (8 bit).
•The upper nibble of this byte (bits 7-4) is called “opcode” and it defines the instruction.
•The lower nibble (bits 3-0) can be used to pass parameters to the instruction (for example a RAM Address).
•Some instructions use lower nibble, they are called memory reference instruction.
•But some other instructions don’t use lower nibble (we will see examples).
•We need to implement 5 Instructions (as is part of SAP-1 architecture): LDA, ADD, SUB, OUT, HLT.

Instruction: LDA

Instruction →	**Load Accumulator**
Mnemonic	LDA
Opcode	0000

Function:

Loads the value stored in a particular RAM address which is specified by the address field of the instruction, then stores it in the accumulator.

Example: LDA 9H
In Binary: 0000 1001

The LDA (Load Accumulator), ADD (Addition), SUB (Subtraction), all of these instructions, are memory reference instructions, meaning that the lower nibble of the instruction set, is NOT ignored, and used.

1. ROM Module Interface (Architectural Intent)

```
module memory(
    input  wire [3:0] addr,
    input  wire       ROM_LOW_OE,
    output reg  [7:0] data
);
```

Explanation:
Defines a **16-address (4-bit) instruction/data ROM** with an **active-low output enable**, consistent with SAP-1's program counter width and shared bus design.

---

2. Combinational ROM Behavior

```
always @(*) begin
```

Explanation:
The `@(*)` sensitivity list ensures **purely combinational behavior**, meaning the ROM output updates immediately with address or enable changes—no clock dependency, as expected for instruction memory in SAP-1.

---

3. Tri-State Bus Control (Output Enable)

```
if (ROM_LOW_OE)
    data = 8'hZZ;
```

Explanation:
When the ROM output is disabled, the data bus enters a **high-impedance (Z) state**, preventing bus contention and allowing other components (e.g., registers, ALU) to drive the bus.

---

4. Address-Based Instruction Fetch

```
case (addr)
    4'h0: data = 8'b0000_1000; // LDA 8
    4'h1: data = 8'b0001_1001; // ADD 9
```

```
    4'h2: data = 8'b1110_1110; // OUT
    4'h3: data = 8'b1111_1111; // HLT
```

Explanation:
Implements **hard-coded instruction storage**. Each address corresponds to a fixed 8-bit instruction, modeling how the **Program Counter fetches instructions** in SAP-1.

---

5. Operand Storage Within ROM

```
4'h8: data = 8'b0000_1001; // Operand: 9
4'h9: data = 8'b0000_1000; // Operand: 8
```

Explanation:
Data operands are stored alongside instructions, reflecting SAP-1's **unified program/data memory** approach and enabling instructions like `LDA` and `ADD` to reference memory directly.

---

6. Safe Defaults and Robust Design

```
default: data = 8'h00;
```

Explanation:
Ensures **deterministic behavior** for undefined addresses, a good HDL practice that avoids simulation artifacts and unintended latches.

---

7. Why ROM (Not RAM) — Code-Level Justification

```
// No write-enable signal
// No data input port
// No clocked always @(posedge clk)
```

Explanation:
The absence of write logic confirms that the module is **intentionally read-only**, matching SAP-1's lack of STORE instructions and reinforcing architectural simplicity.