Lecture 2-3: February 12, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

0.1 Basics of Computer Architecture

Computer architecture deals with interconnection between functional units of m/c.

The age old concept was - it could be the computer engineers view towards the machine computer.

On the other hand, in general, computer organization is the assembly language programmers' view towards a computer system.

A programmer is aware of how basic building blocks such as registers, flags are organized as well as the programmer has familiarity with list of valid machine instructions, number of bits the machine can process etc.

Basic architecture of computer is -

1. Harvard type and
2. von Neumann type

In Harvard Mark1, built in 1944, program and data were stored in separate memories (Harvard Architecture, Figure 1).

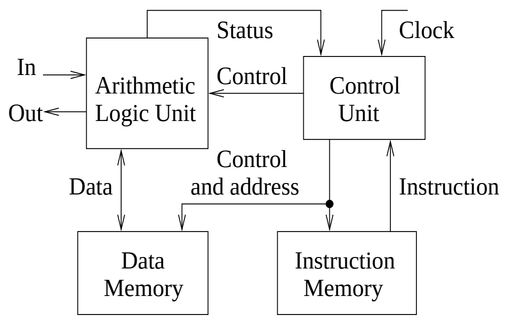


Figure 1: The harvard architecture

0.1.1 von Neumann Architecture

The von Neumann architecture (Figure 2) allows program and data to reside in same memory. This concept is still followed in the modern machine computer.

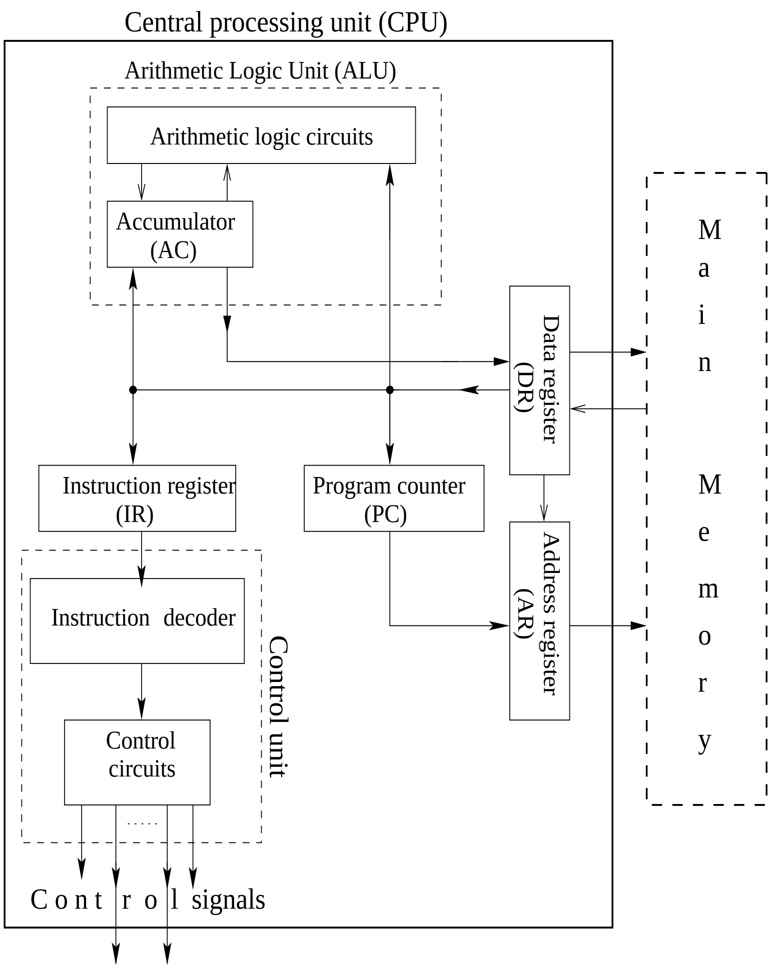


Figure 2: The basic von Neumann architecture

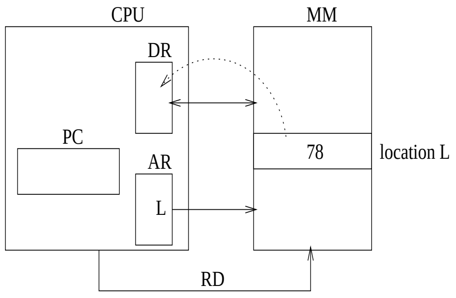


Figure 3: Memory read

- **AR** (address register): While a CPU reads from or writes to a memory location L, the AR of CPU contains the address L.
- **DR** (data register): Also called BR (buffer register).

  A read from memory location L means - data transfer from location L to DR. That is, DR $\leftarrow$ M(AR).

  Write to memory location L implies data available at DR is stored in L.
  That is, M(AR) $\leftarrow$ DR.
- **AC** (accumulator): A special kind of register. It acts as one of the sources as well as destination for most of the CPU arithmetic and logical operations.
- **IR** (instruction register): Contains opcode of an instruction that CPU currently intends to execute.
- **PC** (program counter): Stores next executable instruction address. If CPU currently executes I_i located at memory location L, and next instruction to be executed is at L+1, then PC content during execution of I_i is L+1.

In Neumann's archutecture, there is no explicit distinction between inst and data.

That is,

- An instruction can be treated as data, and can be modified during run time.
- Data can be considered as if it is a valid instruction code, and then can be executed.

  This increases the complexity of debugging an erroneous program.

  The *tag architecture* is realized to protect from any such mishap.

Further, instruction fetch and handling data can't occur at the same time (share common bus) - causes reduced throughput - referred to as von Neumann bottleneck.

0.1.2 The tags

Objective of *tag* is to make the contents of memory location self identifying.

Few extra bits (*tag bits*) are added to each memory word.

In Figure 4, *tag* = 1 (Word 0 and Word 1) implies the word is a data, and *tag* = 0 means the word (Word 2) stores an instruction.

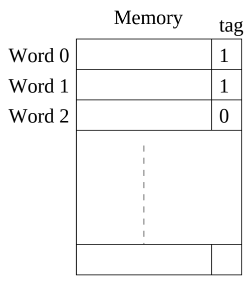


Figure 4: The tag

Tag: Extra cost, additional h/w cost for decoding etc.

Although tag bits increases h/w cost, it reduces the cost towards managing the flow of computation (instruction sequencing),

H/w cost is decreasing but software cost is increasing (last 60 years, Figure 5).

The tags are considered in RISC architecture.

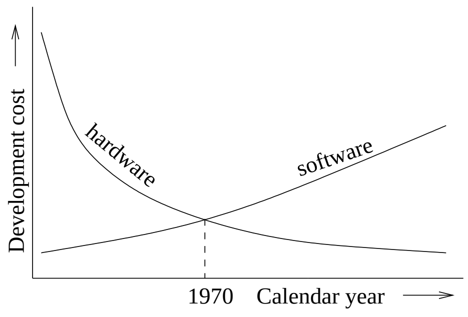


Figure 5: Hardware vs software cost

0.1.3 Instruction sequencing

Option 1: Instruction contains address of next instruction (primitive design/ some modern designs (VLIW).

Option 2: Program counter (PC). This is introduced in von Neumann architecture.

Consider a hypothetical computer with 7 instructions (called macro instructions).

Mnemonic	*Description*	
$LOAD\ X$	$AC \leftarrow M(X)$	$transfers\ content\ of\ location\ X\ to\ AC.$
$STORE\ X$	$M(X) \leftarrow AC$	$transfers\ AC\ content\ to\ location\ X.$
$ADD\ X$	$AC \leftarrow AC + M(X)$	$contents\ of\ location\ X\ and\ AC\ are$ $added\ and\ the\ sum\ is\ stored\ in\ AC.$
$AND\ X$	$AC \leftarrow AC \wedge M(X)$	$contents\ of\ location\ X\ and\ AC\ are$ $anded\ and\ the\ result\ is\ stored\ in\ AC.$
$JUMP\ X$	$PC \leftarrow X$	$unconditional\ branch\ to\ location\ X.$
$JUMPZ\ X$	$if\ AC = 0,\ then\ PC \leftarrow X$	$branch\ to\ location\ X\ if\ AC = 0.$
$COMP$	$AC \leftarrow AC'$	$complement\ AC\ and\ store\ it\ to\ AC.$

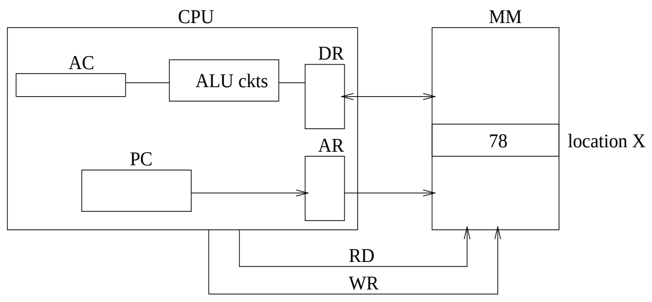


Figure 6: Macro instruction execution

All instructions are 1-addressed i.e, at most one operand is explicitly mentioned.

Other operands of the instruction are implicit.

To execute a macro instruction, CPU computes micro instructions (Figure 7, 8).

$\underline{Mnemonic}$	$\underline{Description}$	
$LOAD\ X$	$AC \leftarrow M(X)$	$transfers\ content\ of\ location\ X\ to\ AC.$
$STORE\ X$	$M(X) \leftarrow AC$	$transfers\ AC\ content\ to\ location\ X.$
$ADD\ X$	$AC \leftarrow AC + M(X)$	$contents\ of\ location\ X\ and\ AC\ are\ added\ and\ the\ sum\ is\ stored\ in\ AC.$
$AND\ X$	$AC \leftarrow AC \wedge M(X)$	$contents\ of\ location\ X\ and\ AC\ are\ anded\ and\ the\ result\ is\ stored\ in\ AC.$
$JUMP\ X$	$PC \leftarrow X$	$unconditional\ branch\ to\ location\ X.$

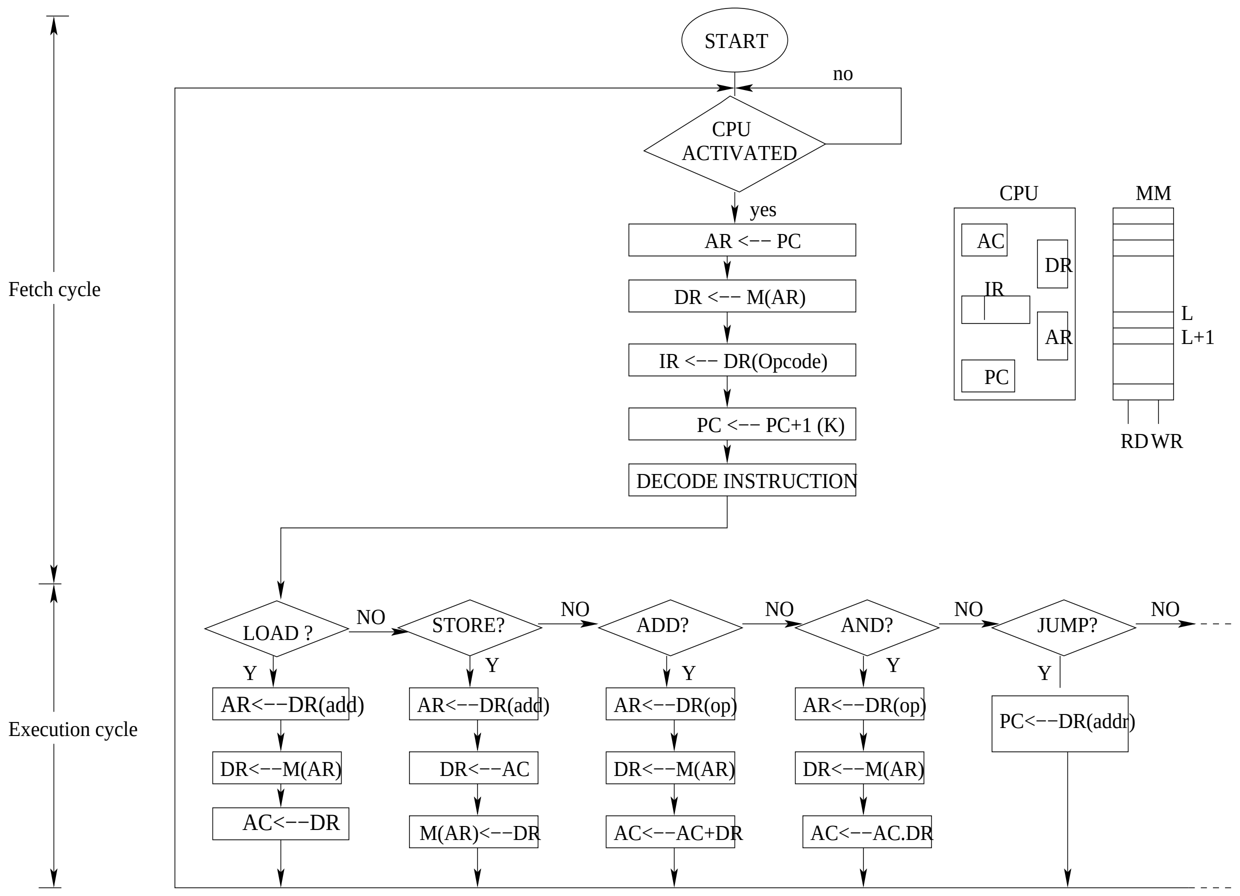


Figure 7: Flow chart

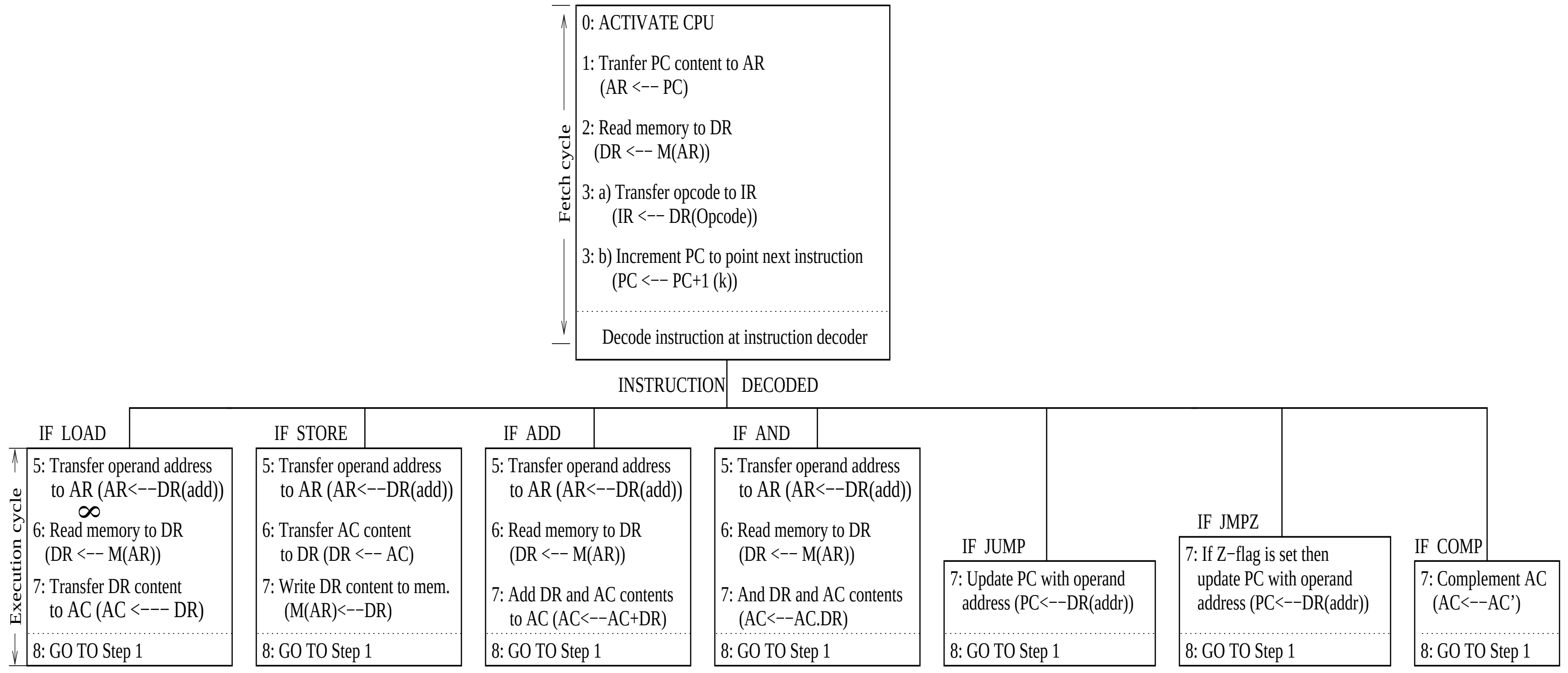

Fetch cycle
0: ACTIVATE CPU
1: Tranfer PC content to AR (AR <-- PC)
2: Read memory to DR (DR <-- M(AR))
3: a) Transfer opcode to IR (IR <-- DR(Opcode))
3: b) Increment PC to point next instruction (PC <-- PC+1 (k))
Decode instruction at instruction decoder
INSTRUCTION DECODED
Execution cycle
IF LOAD
5: Transfer operand address to AR (AR<--DR(add))
6: Read memory to DR (DR <-- M(AR))
7: Transfer DR content to AC (AC <--- DR)
8: GO TO Step 1
IF STORE
5: Transfer operand address to AR (AR<--DR(add))
6: Transfer AC content to DR (DR <-- AC)
7: Write DR content to mem. (M(AR)<--DR)
8: GO TO Step 1
IF ADD
5: Transfer operand address to AR (AR<--DR(add))
6: Read memory to DR (DR <-- M(AR))
7: Add DR and AC contents to AC (AC<--AC+DR)
8: GO TO Step 1
IF AND
5: Transfer operand address to AR (AR<--DR(add))
6: Read memory to DR (DR <-- M(AR))
7: And DR and AC contents (AC<--AC.DR)
8: GO TO Step 1
IF JUMP
7: Update PC with operand address (PC<--DR(addr))
8: GO TO Step 1
IF JMPZ
7: If Z-flag is set then update PC with operand address (PC<--DR(addr))
8: GO TO Step 1
IF COMP
7: Complement AC (AC<--AC')
8: GO TO Step 1

Figure 8: Execution of von Neumann macro instructions

Memory

Memory is one of the major components of computer system (Figure 9).

It includes MM, SM and a high speed component of MM known as cache.

Memory unit with which CPU directly communicates is MM or primary memory.

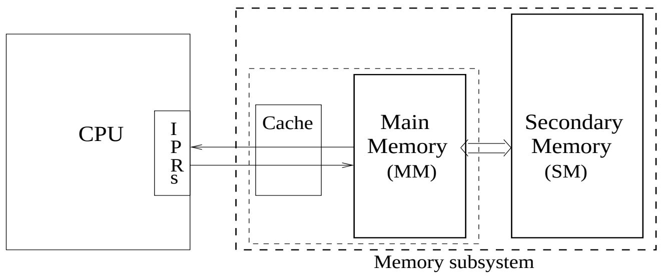


Figure 9: Computer memory subsystem

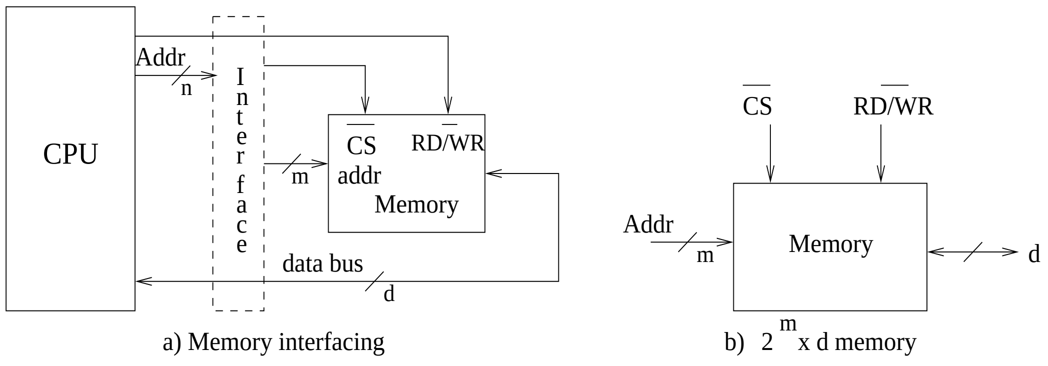


Figure 10: Memory interfacing

0.2 Memory Interfacing

Basic organization of a memory device is shown in Figure 10.

In addition to power supply lines, a memory chip consists of following signal lines.

(i) m address lines to select one out of 2^m memory locations within the chip.

(ii) d bidirectional data lines for data transfer with CPU.

(iii) Read/write signal line(s) to perform read or write opeartion.

$$RD/\overline{WR} = 1 \Rightarrow \text{read from memory}$$
$$RD/\overline{WR} = 0 \Rightarrow \text{write to memory}$$

(iv) At least one chip-select line to enable a chip to be ready for read/write opeartion.

In Figure 10, memory module is having only one active low CS line ($\overline{CS}$).

0.3 Memory Design

Cell is connected to one address driver. If storage capacity is N bits, then it needs one N-output decoder (bit-organized) as well as N address drivers (Figure 11).

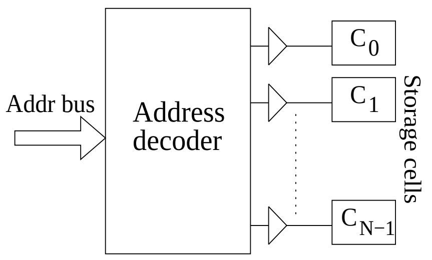


Figure 11: Memory module I

Bit-organized

Memory module: $2^m \times d$ (Figure 12(a)). For bit-organized, $d = 1$ (Figure 12(b)).

Byte-organized For byte organized, $d = 8$ (Figure 12(c)).

Word-organized For word organized, d = word size.

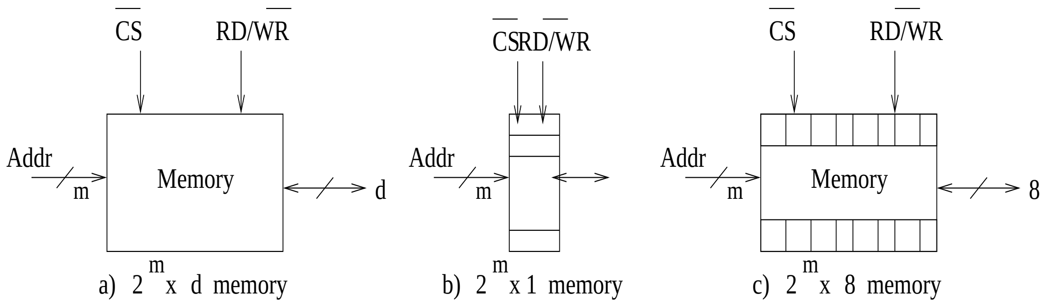
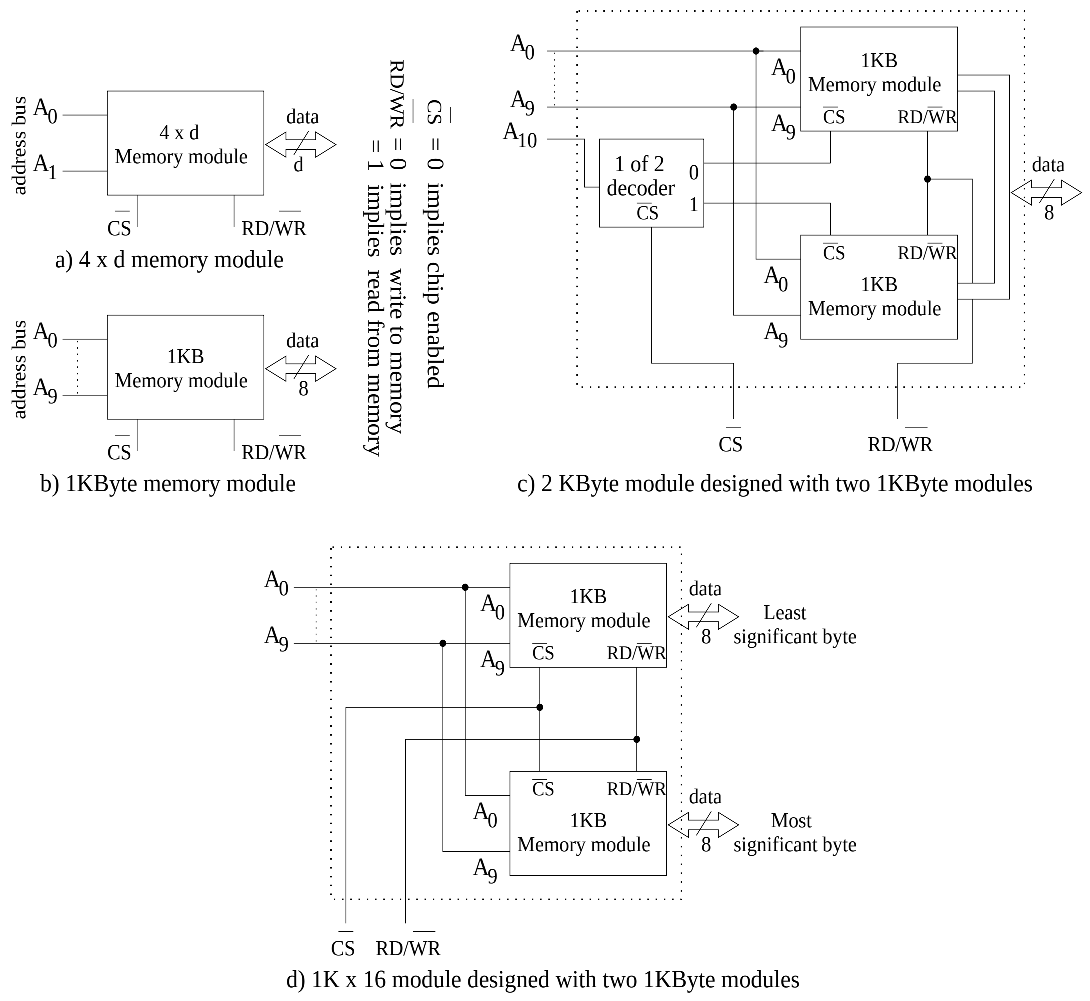


Figure 12: Memory organized

** Design $1\times$ 1 memory

Constructing large memory module

Given $2^m \times d$ memory modules how to design an $2^{m_1} \times d_1$-bit memory module, where $m_1 \geq m$ and $d_1 \geq d$.



address bus
A0
A1
4 x d
Memory module
data
d
CS
RD/WR
a) 4 x d memory module
A0
A9
1KB
Memory module
data
8
b) 1KByte memory module
CS = 0 implies chip enabled
RD/WR = 0 implies write to memory
= 1 implies read from memory
A10
1 of 2
decoder
0
1
c) 2 KByte module designed with two 1KByte modules
Least significant byte
Most significant byte
d) 1K x 16 module designed with two 1KByte modules

Figure 13: Memory arrays

**Design (a) $1\times$ 2 memory, (b) $2\times$ 2 memory

1× 1 memory

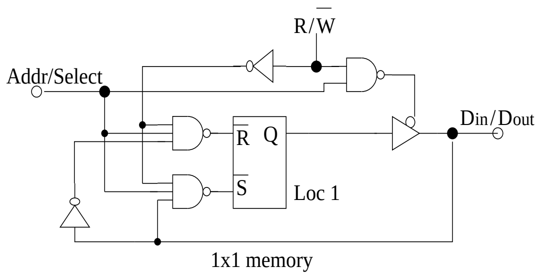


Figure 14: 1x1 memory

Verify the design as per the following table.

Table 1: Verification for 1x1 memory

Sl no	Select	R/$\overline{W}$	Data in [supply]	Data out [verify]	Activity
1	1	0	1	-	Write 1 in Loc-1
2	1	1	-	1	Read 1 from Loc-1
3	0	x	-	-	No operation
4	1	0	0	-	write 0 in Loc-1
5	1	1	-	0	Read 0 from Loc-1

1× 2 memory

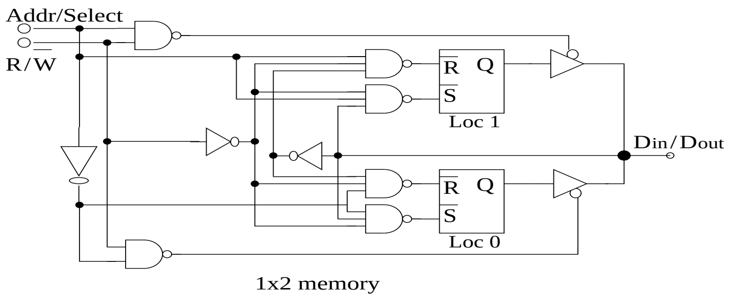


Figure 15: 1x2 memory

Verify the design as per the following table.

Table 2: Verufication for 1x2 memory

No	Select	R/$\overline{W}$	D$_{in}$	D$_{out}$	Verify
1	1	0	1 (d$_1$)	-	Write 1 in Loc-1
2	0	0	1 (d$_2$)	-	Write 1 in Loc-0
3	1	1	-	1 (d$_1$)	Read 1 from Loc-1
4	0	1	-	1 (d$_2$)	Read 1 from Loc-0
5	1	0	0	-	Write 0 in Loc-1
6	0	0	1	-	Write 1 in Loc-0
7	1	1	-	0	Read 0 from Loc-1
8	0	1	-	1	Read 1 from Loc-0
9	1	0	1	-	Write 1 in Loc-1
10	0	0	0	-	Write 0 in Loc-0
11	1	1	-	1	Read 1 from in Loc-1
12	0	1	-	0	Read 0 from Loc-0

2× 2 memory

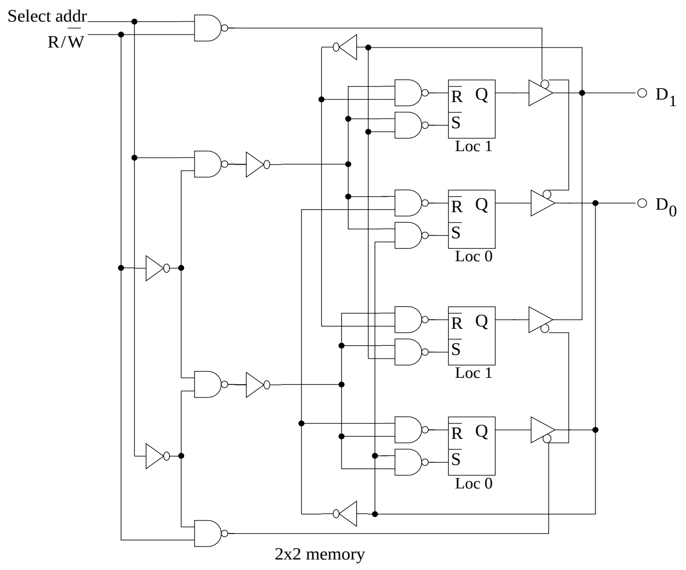


Figure 16: 2x2 memory

Verify the design as per the following table.

Table 3: Verification for 2x2 memory

No	Select	R/$\overline{W}$	D1$_{in}$	D0$_{in}$	D1$_{out}$	D0$_{out}$	Verify
1	1	0	1	0	-	-	Write 10 in Loc-1
2	0	0	0	1	-	-	Write 01 in Loc-0
3	1	1	-	-	1	0	Read 10 from Loc-1
4	0	1	-	-	0	1	Read 01 from Loc-0
5	1	0	0	0	-	-	Write 00 in Loc-1
6	0	0	1	1	-	-	Write 11 in Loc-0
7	1	1	-	-	0	0	Read 00 from Loc-1
8	0	1	-	-	1	1	Read 11 from Loc-0