

Lecture 1: February 9, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

The machine *computer* came in 1940s.

Conceptualized from the Latin word ‘computare’ (‘calculate’ or count), it came to be just as essential to the study of chaotic system.

Study on *computer architecture* is the accumulation of concepts, developed and refined over years - it leads to this powerful computing machine - today’s *Computer*.

0.1 Today’s Computer

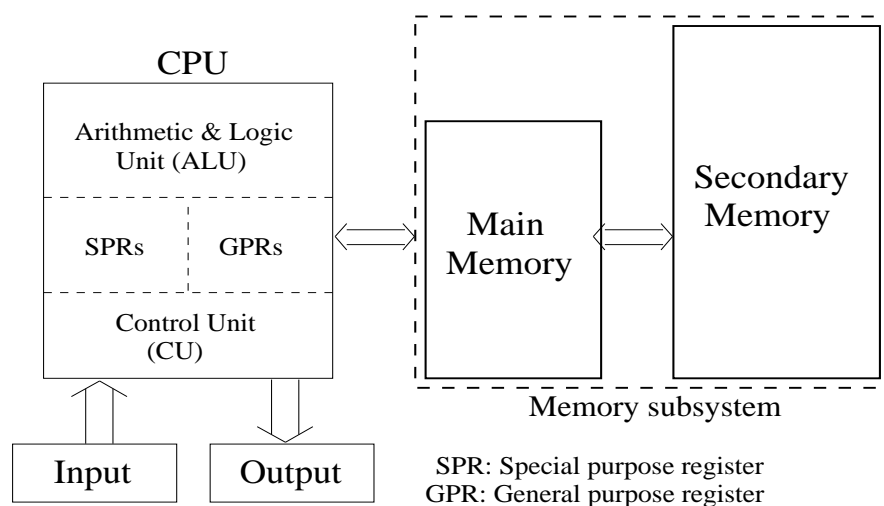
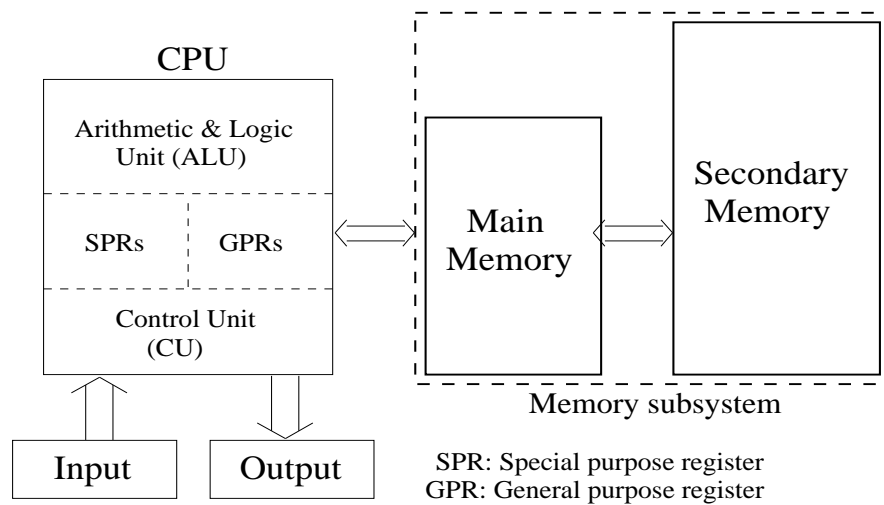


Figure 1: Basic organization of a machine computer

Today’s conventional computer stores the set of instructions in its memory and executes (process) them one by one.

Five major components of a computer (Figure 1) are - **central processing unit (CPU)**, **main memory (MM)**, **input**, **output** and **secondary memory (SM) or backing store**.



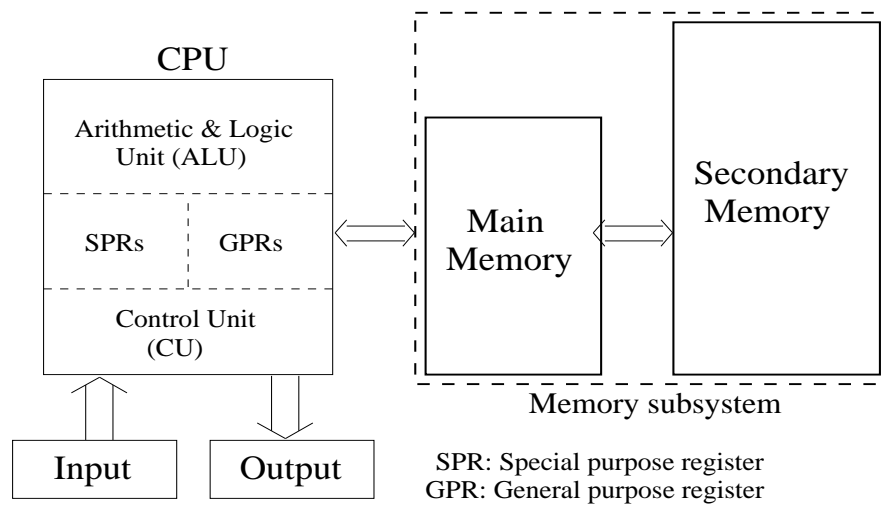
A The central processing unit (CPU)

1. Is the site of all processing.
2. It contains registers

Circuitry to perform arithmetic and logical operations (ALU)

Control circuit (CU) that generates control signals to manage (synchronize) all the operations within a CPU and the machine computer as a whole

In a machine computer, there can be many processors (e.g. Input/Output processors, co-processors, etc.) other than the CPU. However, CPU is the main (central) processing unit of a computer.

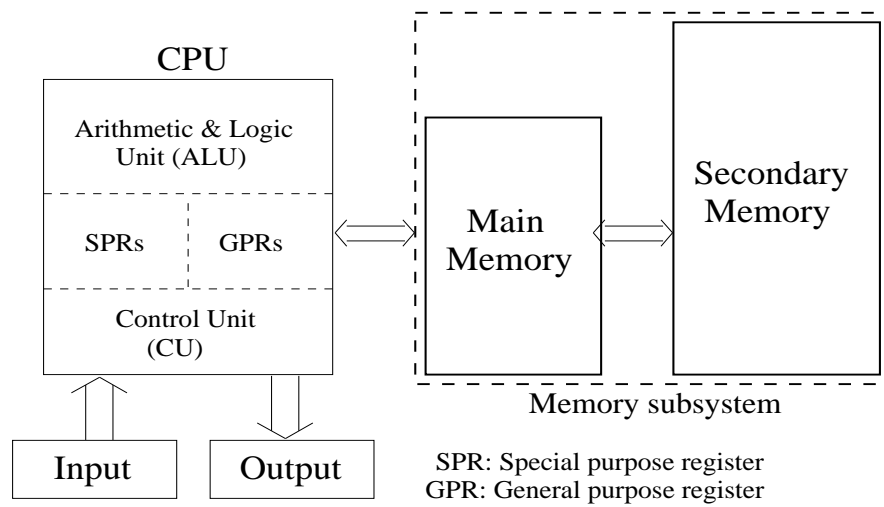


B The main memory (MM)

1. Stores the program (set of instructions) which is to be executed by the CPU.
2. An instruction of a program can not be executed by the CPU unless it is brought into MM which is directly interfaced with the CPU.
3. The major main memory features are-
 - Relatively high speed. Its access time is of the order of nano-second (ns),¹. For example, DRAM speed (access time) is $<50\ ns$.
 - Relatively small capacity, normally, few giga² bytes.
 - Normally, volatile. However, a small portion of MM, is non-volatile ROM. The common example - BIOS ROM.

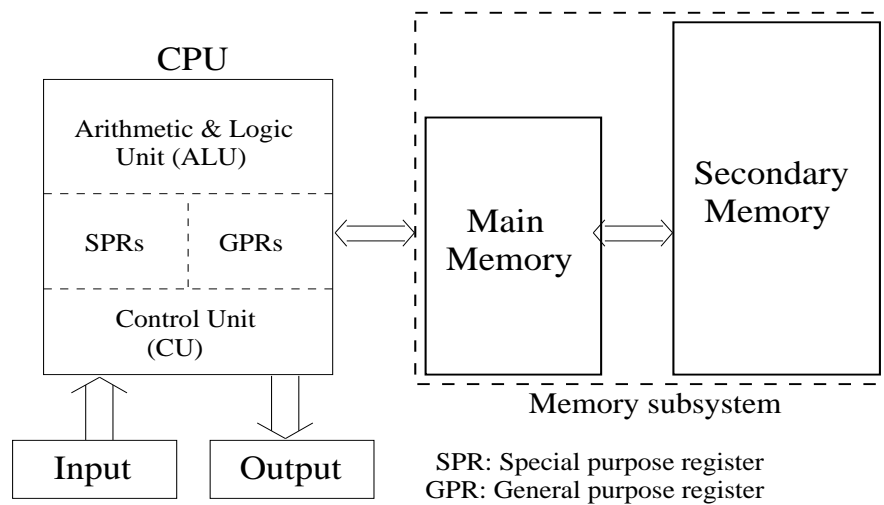
¹ $10^{-3} \Rightarrow$ milli (m), $10^{-6} \Rightarrow$ micro (μ), $10^{-9} \Rightarrow$ nano (n), $10^{-12} \Rightarrow$ pico (p), $10^{-15} \Rightarrow$ femto (f), $10^{-18} \Rightarrow$ atto (a), $10^{-21} \Rightarrow$ zepto (z), and $10^{-24} \Rightarrow$ yocto (y).

²1 giga bytes (GB) = 10^9 bytes.



C The secondary memory (SM)

1. Used for long term storage of program and data.
2. The important features are:
 - Comparatively low speed.
 - Cost/bit is very low.
 - Non-volatile type.
 - Large capacity (even more than a terabyte (10^{12} bytes)).



D Input-output - such devices provide means to establish communication between a user and computer.

1. A wide varieties of input/output devices can be interfaced with the CPU.

These differ as per the information coding style, mechanical/electrical properties, mode of data communication, etc.

Example: communication can follow serial or parallel data transfer; the coding style can be Hollerith/ASCII³, etc.

The speed variation of input/output devices ranges from few bytes to mega bits⁴.

The mode of data transfer between a CPU and the input/output devices follows different options - synchronous, asynchronous, interrupt driven and the direct memory access (DMA).

³ASCII (American Standard Code for Information Interchange) was proposed in 1960.

⁴Keyboard speed can be 0.01 Kbyte/s (10 byte/s). Mouse speed is 0.02 Kbyte/s (20 bytes/s). Modem speed is in Kbyte/s

E Today's computer is nothing but an information processor (Figure 2).

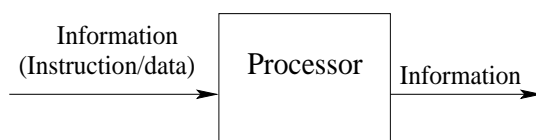


Figure 2: The computer

It can input (accept), compute (process), and output (form) information as desired. Computing (information processing) is an age old practice. It had a long journey to achieve its present shape. We will explore this in AT-I and AT-II.

0.2 The History of Modern Computer

First electronic computer - Electronic Numerical Integrator and Computer (ENIAC, 1946). The ENIAC occupied a space of 30×50 sq ft. It used about 18,000 vacuum tubes (Figure 3).

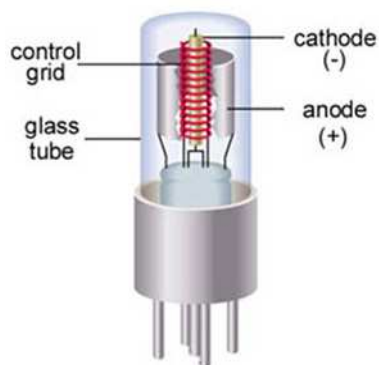


Figure 3: Vacuum tube

Operated (programming/data entry) by switch setting and changing cables.

The system had to switch off every few hours because of over-heating.

The programming in that machine could not be stored for a run.

0.2.1 Stored program computers

Stored program computer is capable of storing programs in its memory.

EDVAC (Electronic Discrete Variable Computer) was the first stored program computer.

Rapid changes in technology and design opened up new directions in the development of machine computer.

For better understanding of the development in electronic computer design, its performance and architecture, we categorize history of its development as generations.

0.3 Computer Generations

History of development of modern electronic computers is categorized based on the device technology, the system architecture, processing mode and language used.

0.3.1 First generation (1940-1955)

ENIAC was introduced in 1946, weighing 30 tons, consisted of 18,000 vacuum tubes, 1,500 relays, 70,000 registers and more than 10,000 capacitors and inductors.

Power consumption of the ENIAC was almost 200 kilowatts.

The area covered by it was 15000 square foot.

ENIAC could compute at most 5000 additions per second.

In 1946-48, von Neumann and his colleagues began the design of a new stored program computer, referred to as IAS (immediate access store) computer.

This effectively set the base of our modern computer architecture.

The important features of first generation machine computers were

- The vacuum tube as basic device technology.
- Relay circuits for realization of the switch.
- The memory, designed around the mercury delay lines.
- Very primitive input/output devices.
- The serial circuitry for computer hardware.
- Programming language was the machine language or assembly language.
- The arithmetic introduced was the bit-serial arithmetic and fixed point.

0.3.2 Second generation (1950-1965)

Second generation computers: transition from vacuum tube to transistor technology.

The transistor was invented in 1947.

Important second generation computer: TRADIC (transistorized digital computer).

TRADIC was made of 800 transistors in 1954.

It was of 3 cubic foot, and could perform million logical operations per second.

TRADIC was operated on 100 kilowatt of power.

Basic features of such second generation computers were

- Transition from vacuum tube to transistor technology.
- Use of magnetic core memory.
- Use of magnetic drum for secondary storage.
- Introduction of HLL such as FORTRAN, ALGOL, COBOL etc.
- Use of index registers in CPU.
- Introduction of floating point arithmetic hardware.
- Introduction of special processors such as I/O processors.
- Batch processing of jobs was in place.

0.3.2.1 Third generation (1960-1975)

Third generation computers introduced the IC technology.

The main features of third generation machine computers were

- Introduction of IC technology. The SSI and MSI were in place.
- Semiconductor memory gradually replaced magnetic core memory.
- Introduction of pipelining architecture.
- Introduction of cache memory in between CPU and main memory.
- Introduction of enriched high level languages.
- Introduction of multiprogramming, time sharing and virtual memory OS.

0.3.3 Fourth generation (1970 - present)

The PCs developed around the microprocessors are the product of fourth generation.

LSI/VLSI technology enables low cost highly efficient portable computers.

Key features of fourth generation computers are

- Extensive use of LSI and VLSI circuits.
- Extension of high level languages to handle both scalar and vector data.
- Introduction of parallel processing techniques for high speed computation.
Massively parallel processors (MPP) are in place.

0.3.4 Fifth generation (1990 - ...)

Fifth generation computer turns to massive number of CPUs for added performance.

Features of fifth generation computers are

- Introduced ULSI/VHSIC (very-high-speed integrated circuits) processors.
- Radical departure from von Neumann architecture.
- Hardware facilities for knowledge processing.
- Design to handle voice and picture input-output.
- Natural language processing.
- Expert computer systems target replacement of expert human brains.

0.4 Technology Trends

VLSI technology made it possible today that the performance of today's microprocessor is even comparable with that of a supercomputer (Figure 4).

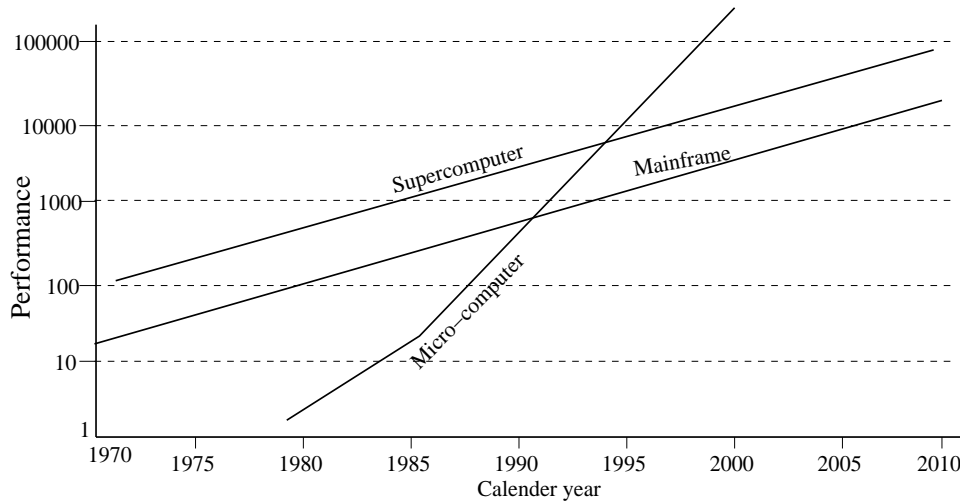


Figure 4: Performance trend

It is due to the improved compaction density within a chip (processor/memory etc.). The figures in Table 1 show the compaction density of different IC technologies.

Table 1: VLSI compaction density

Technology	Type	Compaction density
SSI	TTL	10 gates/chip
MSI	TTL	100 gates/chip
LSI	TTL	1000 gates/chip
VLSI	MOS	10,000 to 1,00,000 gates/chip
ULSI	MOS	Million transistors/chip

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).

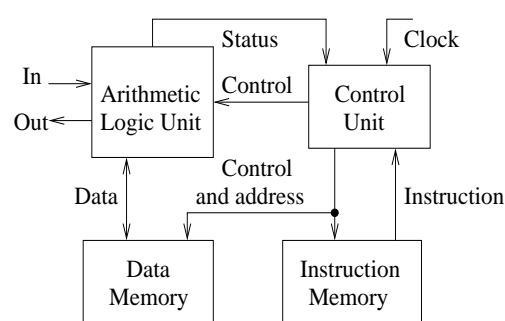


Figure 1: The harvard architecture

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.

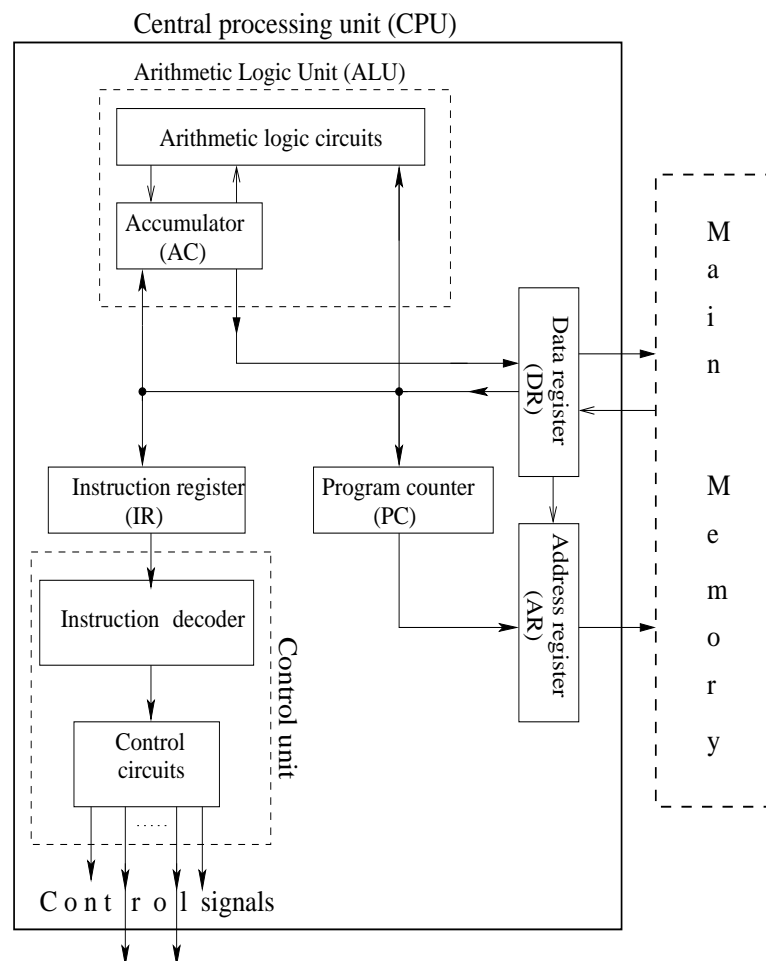


Figure 2: The basic von Neumann architecture

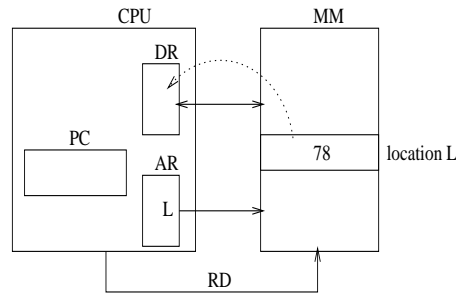


Figure 3: Memory read

- **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 architecture, 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.

	Memory	tag
Word 0		1
Word 1		1
Word 2		0

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.

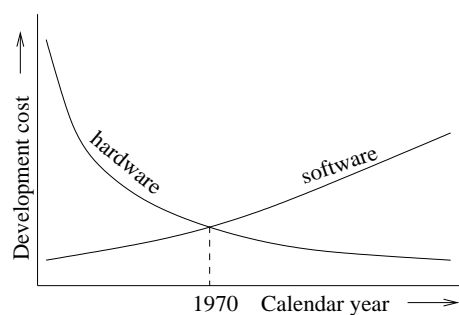


Figure 5: Hardware vs software cost

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).

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

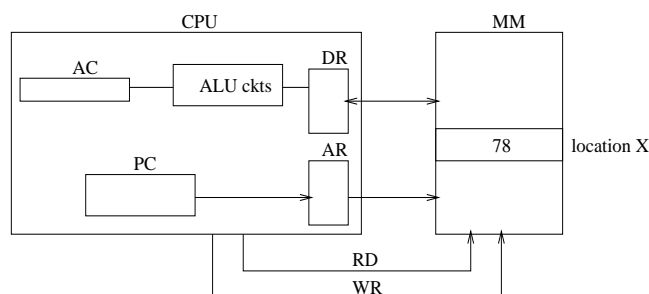


Figure 6: Macro instruction execution

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).

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

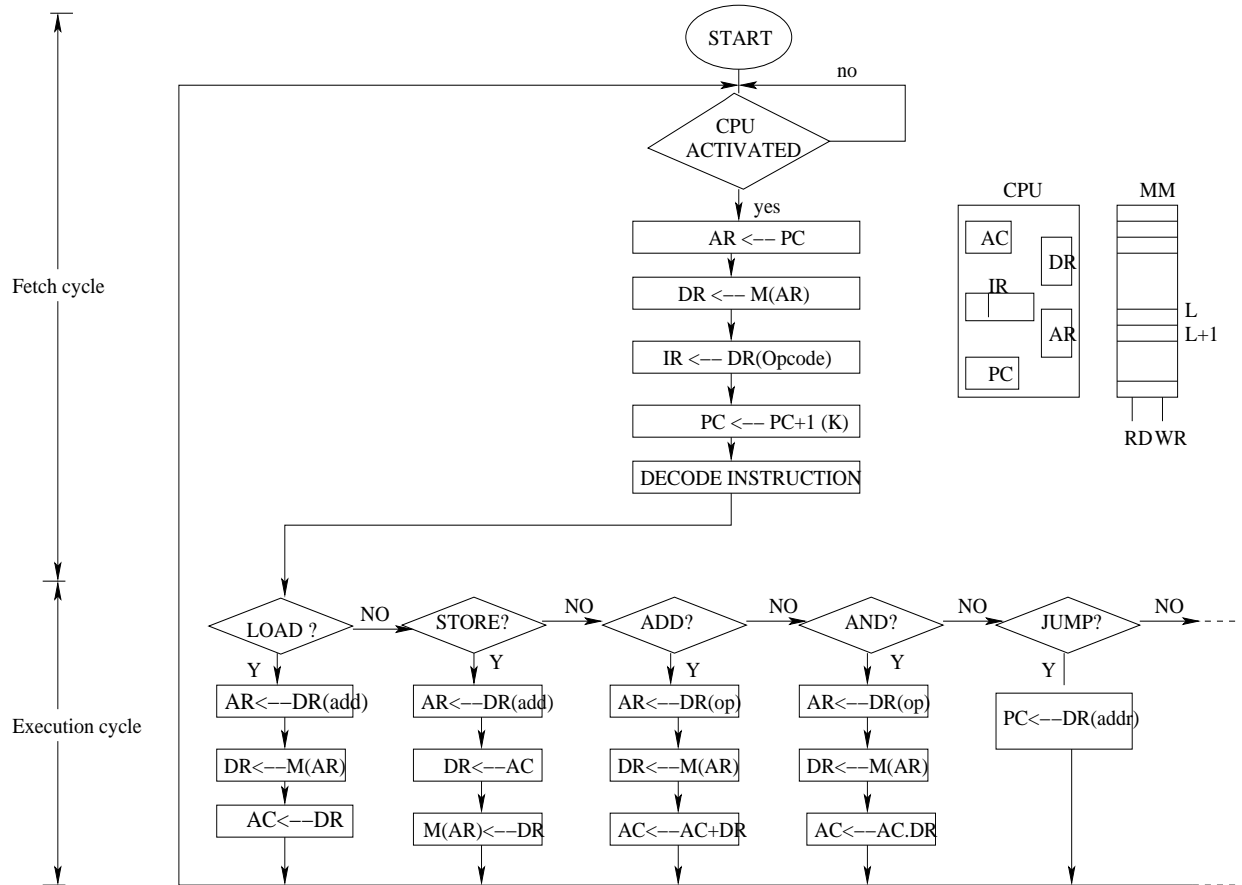


Figure 7: Flow chart

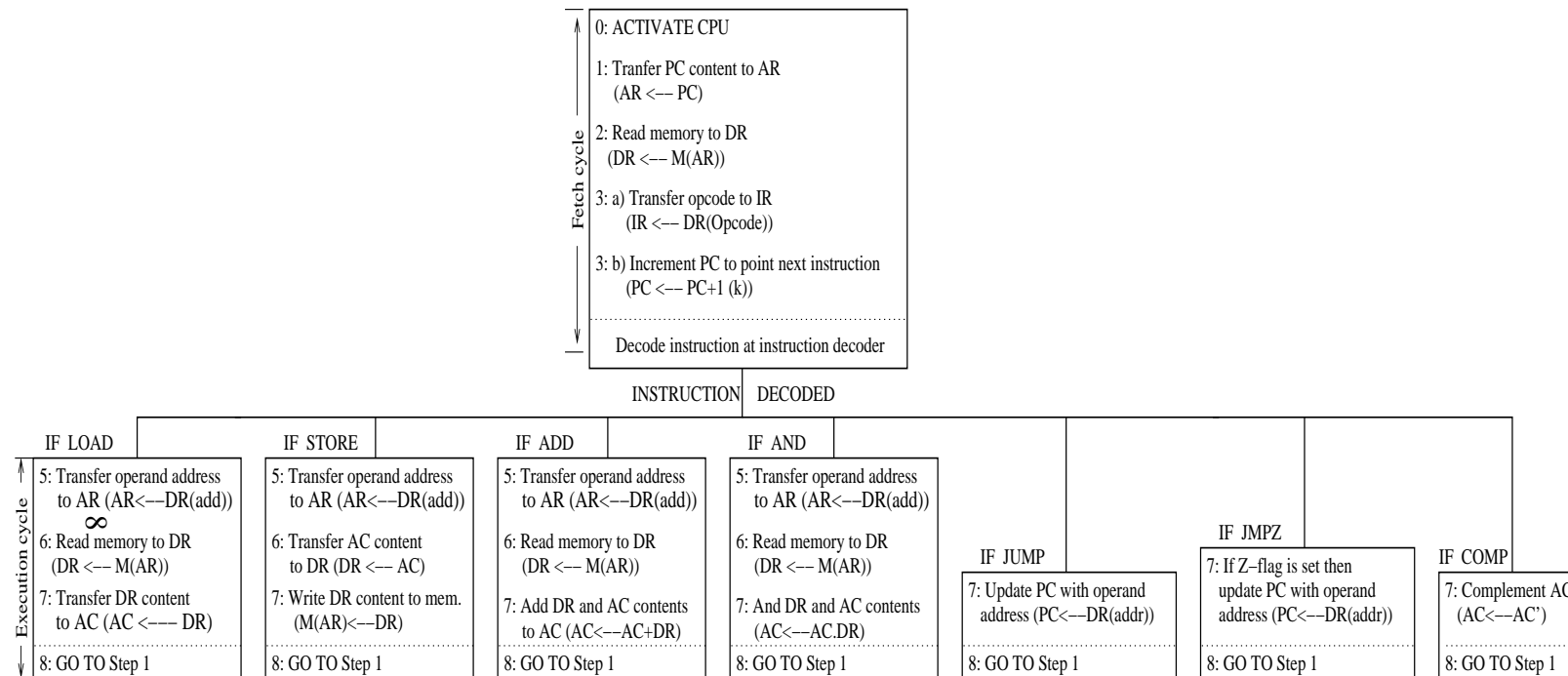


Figure 8: Execution of von Neumann macro instructions

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.

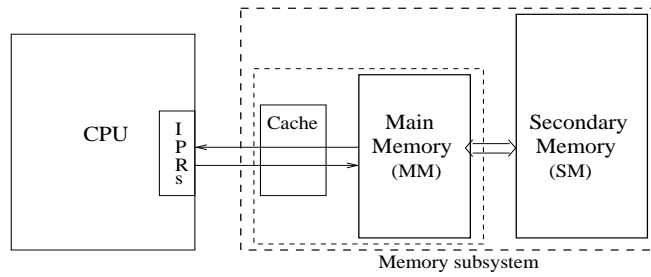


Figure 9: Computer memory subsystem

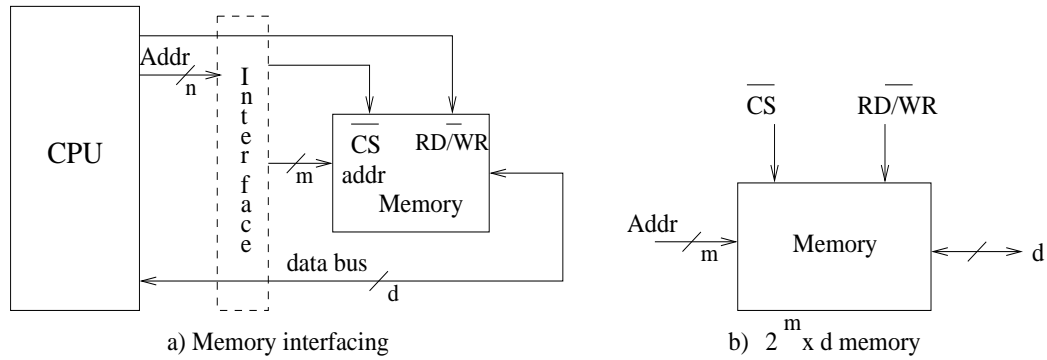


Figure 10: Memory interfacing

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 operation.

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

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

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

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).

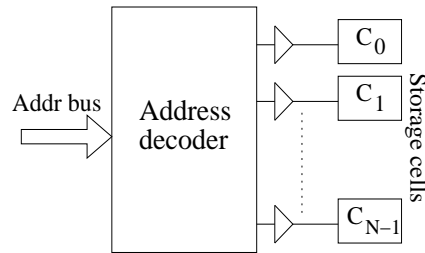


Figure 11: Memory module I

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, $d = \text{word size}$.

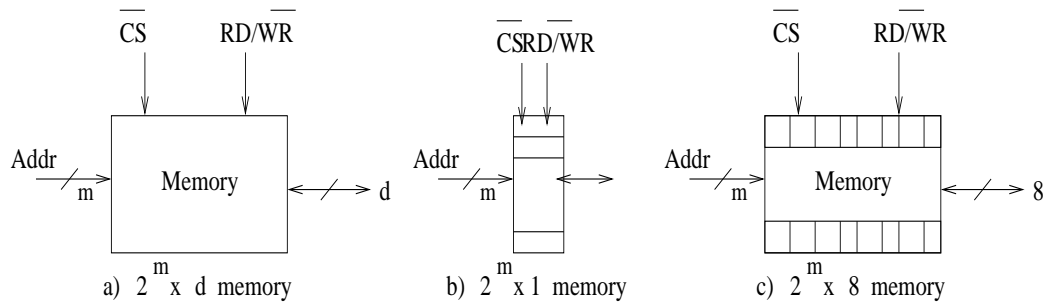


Figure 12: Memory organized

** Design 1×1 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 $d_1 \geq d$.

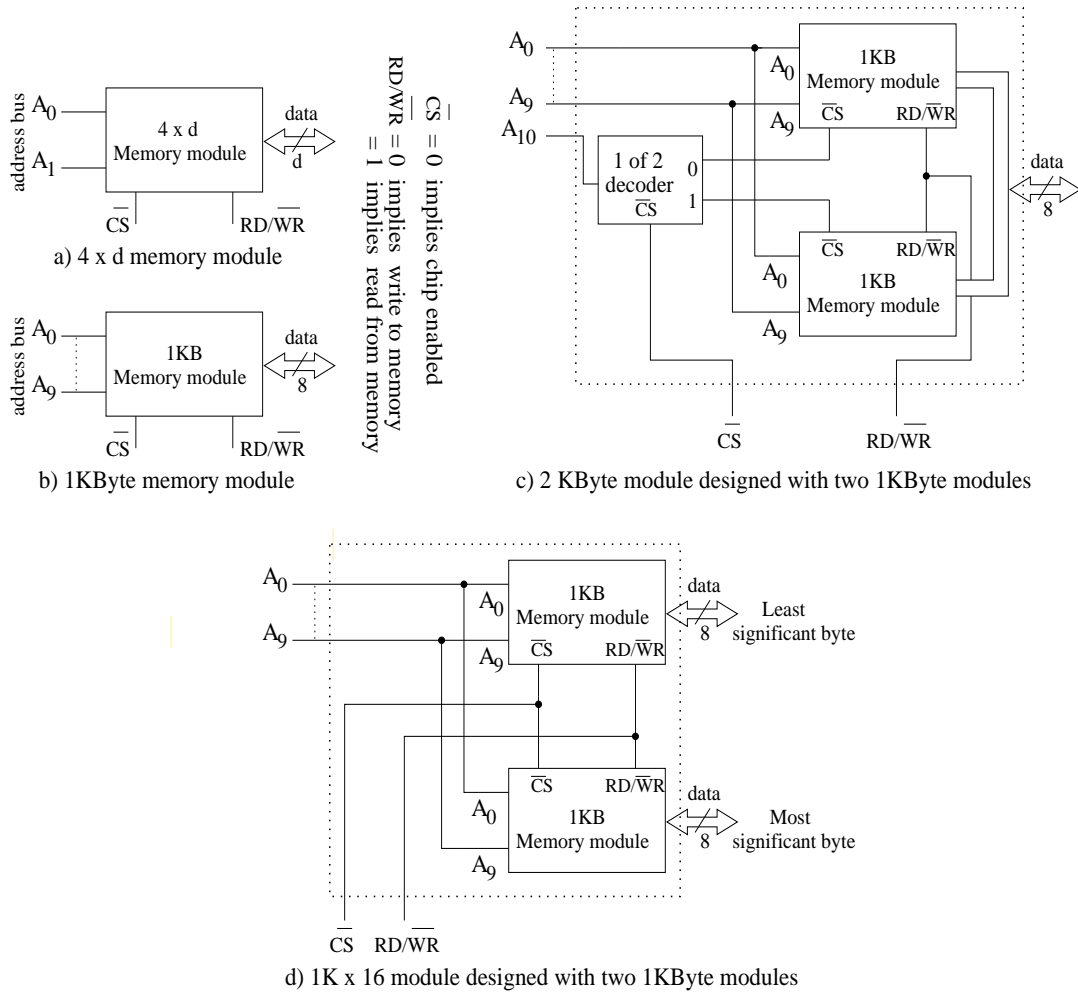


Figure 13: Memory arrays

**Design (a) 1×2 memory, (b) 2×2 memory

1 × 1 memory

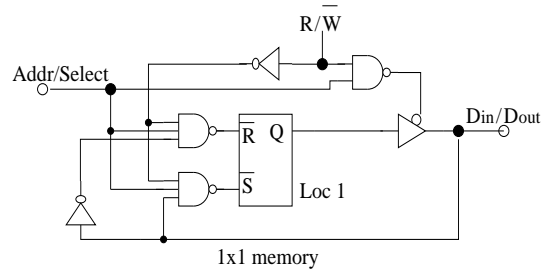


Figure 14: 1x1 memory

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

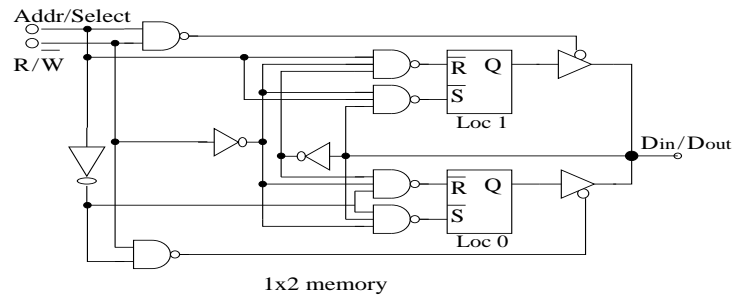


Figure 15: 1x2 memory

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

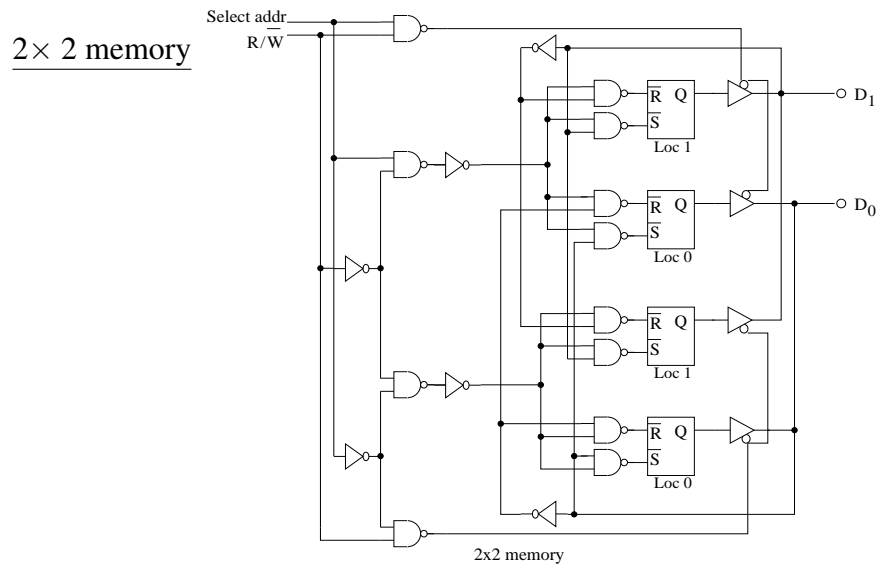


Figure 16: 2x2 memory

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

Lecture 4-5: February 19, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

0.0.1 Extension of basic organization

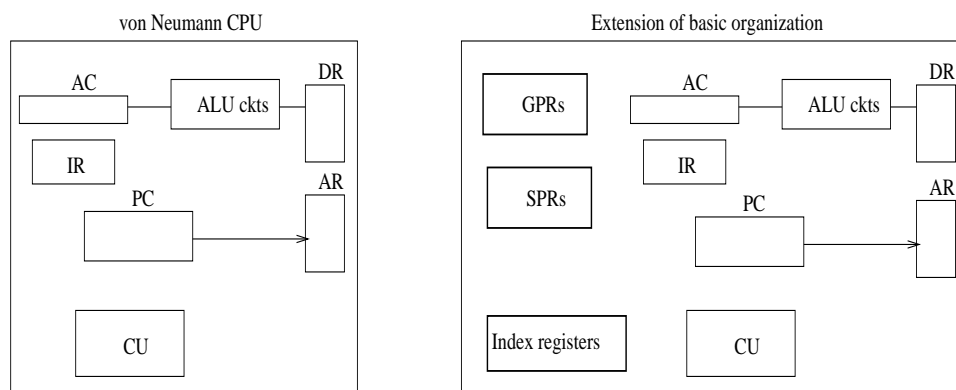


Figure 1: Extension of basic von Neumann architecture

The basic von Neumann architecture is made more powerful by several ways.

1. Additional registers are added. These are:

- **General purpose (multi purpose) registers.** This can be viewed as the replacement of single accumulator by a set of registers. These are explicitly addressable by the processor through instructions.
- **Index registers.** While defining operands in an instruction the instruction may also specify the index address. [an index register is a type of processor register used to store an offset or index value used in memory addressing](#)
The content of index register is then added to the operand address specified in instruction to get the (effective) address of operand.

???? • **Special purpose registers** - SP, segment registers, status register etc.

2. Capabilities of the ALU are enhanced. In basic von Neumann architecture, simple arithmetic was introduced. In the modern design,

- The multiplication, division and other complex instructions are considered.
- Floating point hardware is introduced.

3. Instruction prefetching is introduced. Consider execution of I, I+1, I+2, Concept of basic von Neumann architecture is- fetch I from memory and execute. I+1 is fetched only after completion of I. In instruction prefetching, I+1, I+2, ... , that are otherwise ready for execution, are fetched while I is in execution (Fig. 2).

Instruction prefetching works by predicting the instructions that are likely to be executed next and fetching them from memory before they are actually needed.

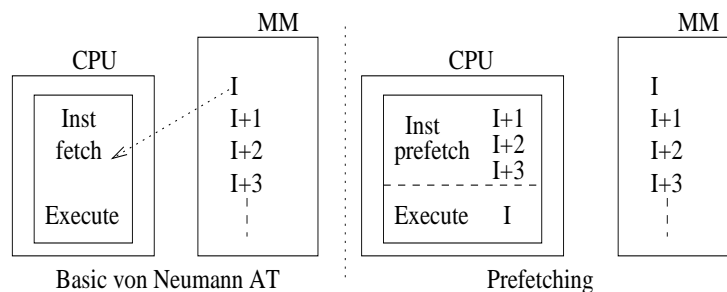


Figure 2: Instruction prefetching

4. **Special control circuitry added:** Initial proposal was a single CU(). At present, CPU can have multiple control units. For example, in addition to main CU (called supervisor CU) CPU is having multiplier CU (the slave unit, Figure 3).

These circuits are typically integrated into the central processing unit (CPU) or other system components, and are responsible for managing the operation of the system at a low level.

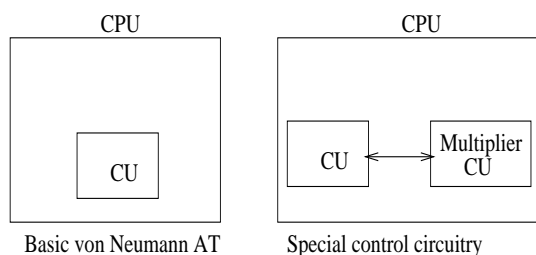


Figure 3: Special control circuitry

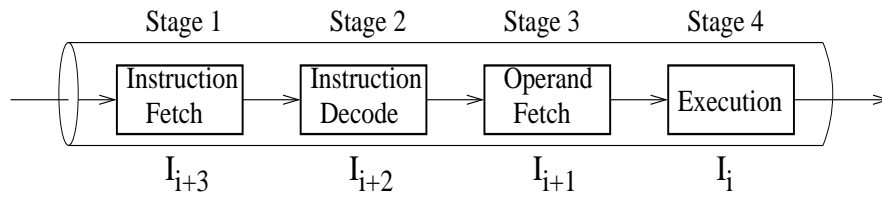


Figure 4: Pipelining

5. **Speed up through pipelining within the CPU** (Figure 4).

6. **Parallel processing**: Simultaneous processing of two or more distinct instructions. **Several instructions and/or operands are fetched simultaneously.**

Execution of more than one instruction is overlapped. The options are

- ALU is divided into K-parts (Figure 5) - K partitioned ALU. It allows simultaneous execution of one integer instruction, a floating point instruction, one logical operation etc.

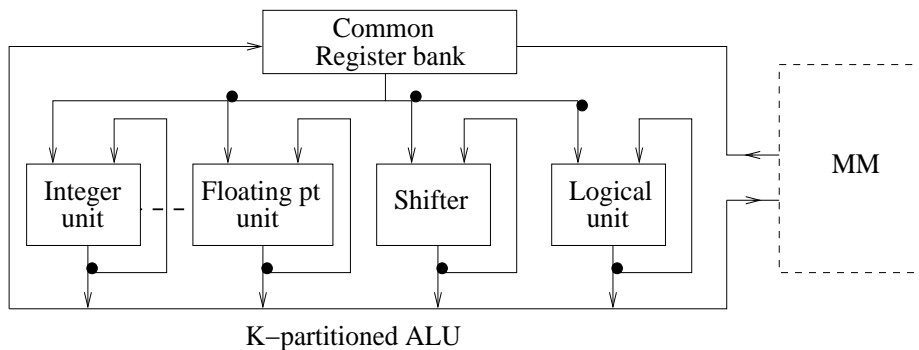


Figure 5: CPU with K-partitioned ALU

- ALU is replicated K-times (Figure 6). The K copies of an ALU circuit allow simultaneous execution of K instructions.

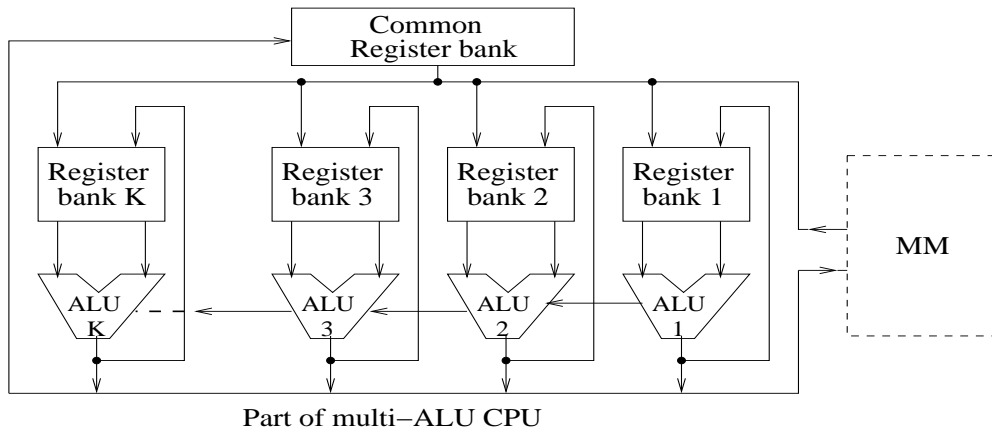


Figure 6: CPU with replicated ALUs

Latest introductions in parallel processing architecture are - multiple processor core, very large instruction word (VLIW) architecture, superscalar architecture etc.

7. Speculative execution:

Speculative execution is a technique used in modern processors to improve their performance by allowing them to execute instructions before it is certain that they will be needed.

8. Green computing:

Green computing is the practice of designing, developing, and using computer systems and technology in an environmentally sustainable way.

Instruction Set Architecture

The main function of the CPU is to execute user program stored in memory.

CPU can only directly fetch instruction of a program from MM.

The instruction is then decoded, operands of instruction (if there is any) is fetched, and finally the execution of instruction is done by CPU.

CPU can execute only valid instructions (called macro instruction) specified in its instruction set.

a macro instruction is a single statement or command that expands into a set of instructions or statements, often used to simplify programming tasks and reduce code duplication.

Instruction set architecture of a CPU refers to programmer visible instruction set.

The type of instructions that should be included in a general purpose processor's instruction set can not be arbitrary.

The instruction set architecture design follows a few guidelines.

0.1 Instruction Set Design

Following requirements are to be fulfilled while defining instruction set of a m/c.

1. Complete: It means - a programmer should be able to construct machine language program to solve most of the common problems.

2. Efficient: Instruction set of a computer is to be such that - instructions used most frequently by the programmers take less time to execute.

That is, instructions like simple data movement, integer arithmetic instructions, simple logical instructions, branch control instructions etc. should take less time for execution.

3. Regular: If an instruction I_T performs task T, there should be an instruction I_U that can undo the task T.

For example, if an instruction I_{Lshift} is included to perform one bit left shift of accumulator, then there must be an I_{Rshift} instruction that can perform one bit right shift on accumulator.

Similarly, if INR (increment content of a register) is considered, then to be regular DCR must be included.

4. **Compatible:** To reduce hardware and software costs, an instruction set must be compatible with existing machine instructions.

It ensures - design cost for implementing the instructions becomes affordable.

Example: Say, the design for 16-bit addition logic is available. Then inclusion of 32-bit addition instruction can be affordable. However, the choice of 24-bit addition instruction may result in exorbitant implementation cost.

Generic instructions types conventionally included in a computer instruction set

i) **Data transfer** (data movement) instructions: Example - Move, Load, Store, Swap, Push, Pop, Clear, Set etc.

ii) **Arithmetic instructions:** Example- Add, Sub, Multiply, Division, Increment, Decrement, Negate, Absolute etc.

iii) **Logical instructions:** Example- Set, Reset, Rotate, And, Or, Ex-or, Not, Shift, Translate, Convert, Edit etc.

iv) **Program control instructions:** Example- Execute, Skip, Jump, Test, Compare, Set control variables, Wait, Nop, Call, Return, Halt etc.

v) **Input/output instructions:** Example- Read, Write, Start I/O, Halt I/O, Test I/O or channel etc.

vi) **Special purpose instructions:** Example- Diagnose, Trap, Breakpoint, instructions for operating system etc.

0.2 Instruction Format

Follow Figure 7.

Opcodes are typically represented as a binary code, consisting of a fixed number of bits that specify the operation to be performed and any associated operands or parameters

In computer programming, opcodes (short for "operation codes") are instructions that specify the operations to be performed by a processor or microcontroller.

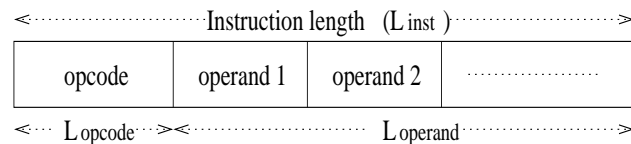


Figure 7: Instruction format

There can be zero or more operands - explicitly specified.

L_{inst} is the instruction length ($L_{opcode} + L_{operand}$).

Memory requirement to store a program depends on the length of individual instruction as well as the number instructions needed.

A main memory capable of delivering N bits/s can make N/L_{inst} number of instructions/s available to the CPU.

A CPU is having shorter instruction length can access larger number of instructions per cycle leading to a faster program execution.

Further, instruction length has the effect on size of instruction decoder and the CPU registers such as DR, IR etc.

Normally, widths of CPU data paths are same as that of word size/half-word size/byte. Therefore, CPU instruction length of one word/half-word/byte is desirable.

Optimization of instruction length follows two options

- Optimization of the length of operand field, and
- Optimization of the opcode length.

0.3 Operands Optimization

Length of operand field is optimized by reducing the explicit operands as well as through optimization of the length of individual operand field.

Example: in a processor if there are 32 general purpose registers, the instruction

$$\text{ADD } R_i \text{ (AC} \leftarrow \text{AC} + R_i \text{)}$$

requires 5 bits to specify R_i mentioned explicitly. On the other hand, 10 bits are required for specifying operands of

$$\text{ADD } R_i, R_j \text{ (} R_i \leftarrow R_i + R_j \text{)}$$

Here, the two operands R_i and R_j are mentioned explicitly.

0.3.1 m -addressed machine

Number of operands explicitly specified in an instruction is reduced.

For an operation with n operands, only m of them are explicitly mentioned in the instruction format, where $m \leq n$. The rest $n-m$ operands are implicit.

For example, $\text{ADD } X \text{ (AC} \leftarrow \text{AC} + M(X))$ specifies only one operand (X) explicitly.

However, implementation of such an instruction requires 3 operands.

The implicit other two operands are the AC (accumulator).

Conventionally instruction set are designed considering $m = 0/1/2/3$.

Small m signifies reduced size of an instruction.

However, if number of explicit operands (m) is reduced, the number of instructions needed to get solution to a problem increases.

Therefore, choice of an optimum m is to be decided.

Definition 0.1 A computer is called an m -addressed machine if each of the instructions in its instruction set is p -address, where $p \leq m$ and $m = 1, 2, \dots$.

Definition 0.2 In a zero-addressed machine ($m = 0$), all the instructions except *PUSH* and *POP* are 0-address.

0.3.1.1 3-address instruction

Here, number of explicit operands in an instruction is $m=3$. Instruction of the form

$$\text{ADD A, B, C} \Rightarrow A \leftarrow B + C$$

is an example of 3-addressed instruction.

0.3.1.2 2-address instruction

Example, in the following 2-addressed instruction

$$\text{ADD A, B} \Rightarrow A \leftarrow A + B,$$

the A represents two operands - one explicitly specified and other one is implicit.

0.3.1.3 Single-address instruction

In one-address (or single-address) instruction, only one operand is explicitly specified in the instruction representation.

All other operands of the instruction are implicit.

In the single-address instruction

$$\text{ADD B} \Rightarrow AC \leftarrow AC + B,$$

the other two operands are the accumulator (AC).

0.3.1.4 Zero-address instruction

In a zero-address instruction representation, all the operands are implicit. Example:

$$\text{ADD} \Rightarrow \text{STACK}[\text{Top}] \leftarrow \text{STACK}[\text{Top}] + \text{STACK}[\text{Top}-1].$$

Two top elements of STACK are added and result is placed on the top of STACK.

The zero-addressed m/c may not be the best choice.

Though instruction length decreases, the program size increases in terms of number of instructions required to write the solution to a problem.

Example 0.1 Consider evaluation of $Z = w - x + y$.

The solution in 3-addressed m/c requires 2 instructions.

$$\begin{aligned}\text{SUB } Z, w, x &\Rightarrow Z \leftarrow w - x \\ \text{ADD } Z, Z, y &\Rightarrow Z \leftarrow Z + y\end{aligned}$$

The solution in 2-addressed m/c requires 3 instructions.

$$\begin{aligned}\text{ASS } Z, w &\Rightarrow Z \leftarrow w \\ \text{SUB } Z, x &\Rightarrow Z \leftarrow Z - x \\ \text{ADD } Z, y &\Rightarrow Z \leftarrow Z + y\end{aligned}$$

In 1-addressed m/c, the number of instructions is 4.

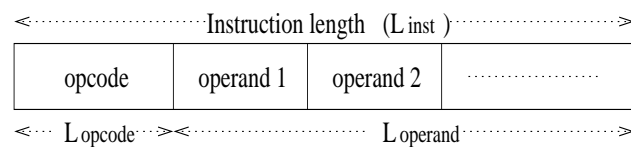
$$\begin{aligned}\text{LOAD } w &\Rightarrow \text{AC} \leftarrow w \\ \text{SUB } x &\Rightarrow \text{AC} \leftarrow \text{AC} - x \\ \text{ADD } y &\Rightarrow \text{AC} \leftarrow \text{AC} + y \\ \text{STORE } Z &\Rightarrow Z \leftarrow \text{AC}\end{aligned}$$

The solution in 0-addressed m/c

$$\begin{aligned}&\text{PUSH } w \\&\text{PUSH } x \\&\text{SUB} \\&\text{PUSH } y \\&\text{ADD} \\&\text{POP } Z\end{aligned}$$

requires 6 instructions.

0.3.2 Operand addressing



The technique to refer an operand is known as the operand addressing mode.

0.3.2.1 Immediate addressing

In immediate addressing one or more operands are specified within the instruction operand fields as constant.

It avoids computation of operand address and accessing of registers or memory.

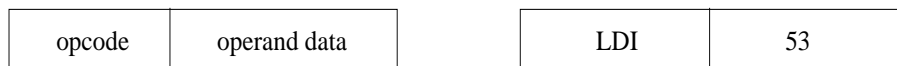


Figure 2: Immediate addressing mode

If word size is smaller than the number of bits required to address an operand, then this scheme can help in reducing the instruction length. In Intel 8085,

LDI 53

transfers 53 to AC. It is a 2-byte instruction (one for LDI and other for 53).

On the other hand,

LDA X

also transfers 53 to AC if location X contains 53.

LDA X is a 3-byte instruction. 1-byte is for LDA and 2 bytes are to specify the operand's address in main memory.

0.3.2.2 Absolute or Direct addressing

It can be memory direct (Figure 3(a))/ register direct (Figure 3(b)).

No computation is required to find the effective address of operand (A_{eff}).

In memory direct addressing only one memory reference is required to get operand.

Size of operand field depends on the size of memory address space.

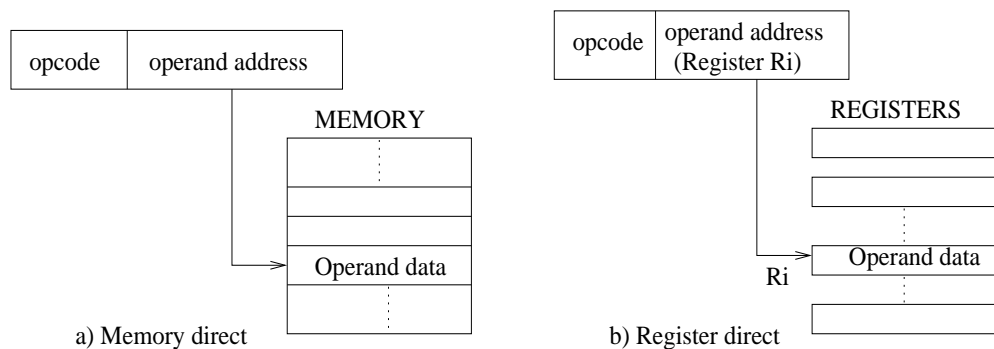


Figure 3: Direct addressing mode

Example of direct addressing -

Add X, Y $\Rightarrow X \leftarrow X + Y$; Memory direct addressing; X, Y memory locations.

Add $R_1, R_2 \Rightarrow R_1 \leftarrow R_1 + R_2$; Register direct addressing; R_1, R_2 registers.

0.3.2.3 Indirect addressing

The address specified in the operand field directs the address of operand.

If operand field is X, the effective address of operand is

$$A_{eff} = [X].$$

X can be a memory location or a register.

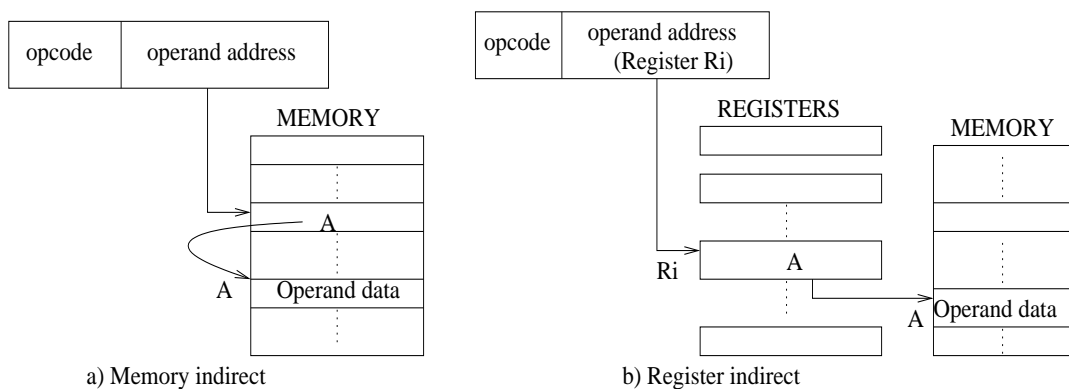


Figure 4: Indirect addressing mode

There can be memory indirect (Figure 4(a)) or register indirect (Figure 4(b)).

Example: memory indirect -

$$\text{ADD I } X \Rightarrow AC \leftarrow AC + [[X]] \Rightarrow \leftarrow AC + [A].$$

For register direct or indirect, the size of address field is very small.

Indirect addressing introduces delay in operand fetching.

0.3.2.4 Relative addressing

In relative addressing, operand field specifies displacement of actual operand from the current instruction in execution (Figure 5) or PC content.

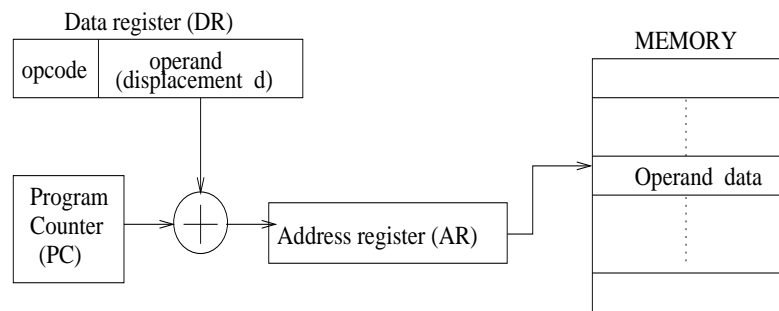


Figure 5: Relative addressing mode

Example:

Store Rel d

Stores content of accumulator (AC) to memory location $L = PC + d$. That is,

$$A_{eff} = PC + \text{displacement } (d).$$

It signifies that a part of operand address is implicit in relative addressing.

Relative addressing avoids relocation of operand addresses while loading a program in memory.

In a program if most of the memory references are close to instruction in execution, then relative addressing can be very effective in reducing the operand field size.

Example: Let CPU is with 16 address lines and program size is less than 1K word. Then in relative addressing, operand field $d=10$ -bit. In direct addressing, it is 16-bit.

0.3.2.5 Page addressing

In this mode, only a part (say YY) of operand address is specified in operand field. Operand address is computed considering content (XX) of page register and YY.

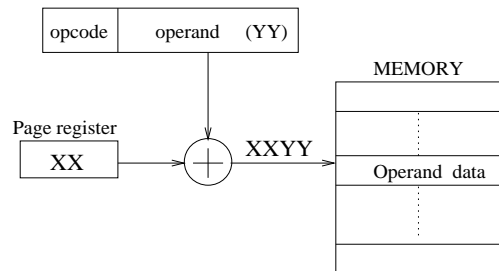


Figure 6: Page addressing

Page register content is modified by OS when switched to different page of memory.

0.3.3 Base addressing

Operand field specifies displacement of operand relative to content of a base register specified in instruction.

Base register reference can also be implicit (not mentioned explicitly in instruction).

There can be a number of base registers in a computer.

Base register is loaded with a value when program is loaded for execution.

It can be effective one when there is locality of memory references.

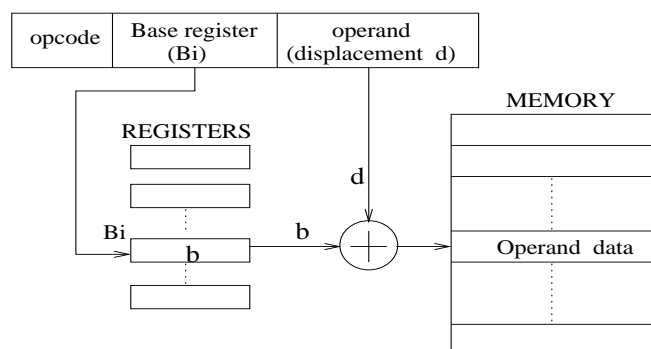


Figure 7: Base addressing

0.3.4 Indexed addressing

In this addressing, operand field of an instruction specifies a memory address A and an index register IX that contains displacement of operand from A . That is,

$$A_{eff} = A + [IX].$$

Example: Load $R_d, 200(I_x) \Rightarrow A_{eff} = 200 + 20 = 220$ if content of I_x is 20.

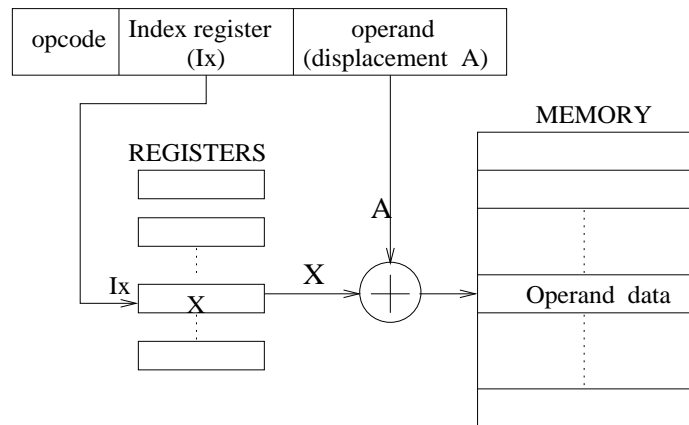


Figure 8: Indexed addressing

Number of bits required for the operand field is comparatively more.

It is effective when we need to perform iterative operations.

Example: In a program if we need to execute the same operation I iteratively on different operands at $A, A+1, A+2, \dots$, then with the help of indexing this can be realized in a simpler way. In indexing,

- The I 's operand field is set to A ,
- The index register (IX) chosen is initialized to 0,
- After each I operation, the IX is incremented by 1.

Some systems realize *autoindexing* to increment/decrement the index register.

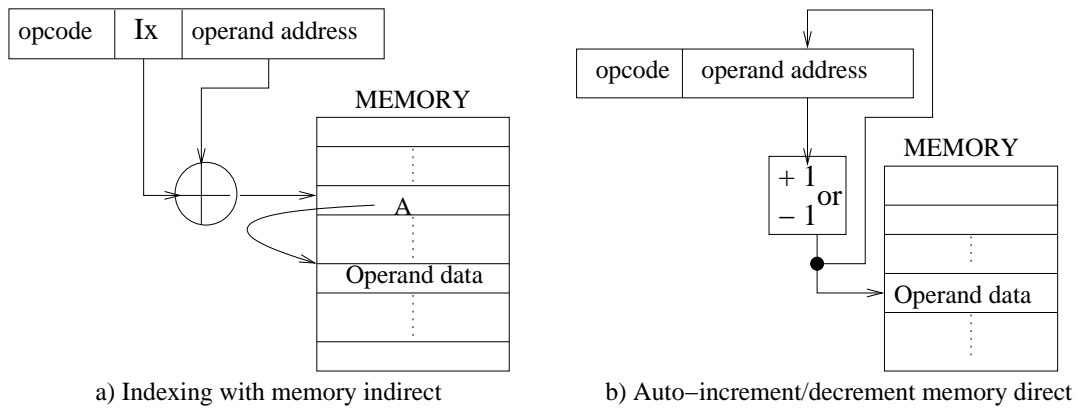


Figure 9: Compound addressing

0.3.5 Compound addressing

There are systems where one or more addressing modes are mixed.

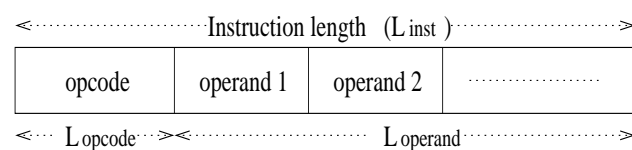
Two examples are in Figure 9.

The x86 allows a variety of addressing modes such as

Immediate addressing, direct addressing, relative addressing, base addressing, a more sophisticated form of index addressing, base with index and displacement etc.

RISCs allow simple/straightforward addressing modes in comparison to CISC.

0.4 Opcode Optimization



Width of opcode field depends on the number of instructions in the instruction set.

Opcode can be of fixed length -that is, fixed number of bits for opcode field.

In fixed format, for n -bit opcode, there can be 2^n instructions in instruction set.

However, reducing n cannot always be a realistic solution.

The alternative: choice of variable length opcode.

Opcode extension was introduced in m/c of old days. Example: DEC PDP-8.

Instruction length is 12-bit (Figure 10).

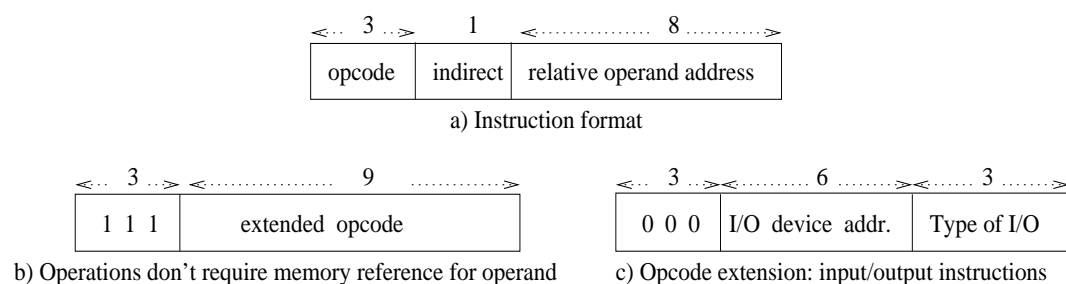
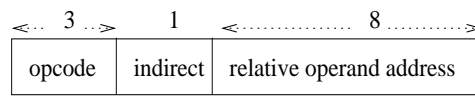
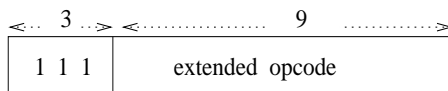


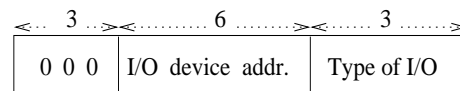
Figure 10: Instruction format of PDP-8



a) Instruction format



b) Operations don't require memory reference for operand



c) Opcode extension: input/output instructions

In normal representation, 3-bit is specified for opcode, 1-bit is for direct or indirect addressing and 8-bit is for relative operand (memory) address.

Out of eight unique patterns in 3-bit opcode (Figure 10(a)), six patterns (001 to 110) are for opcode, each refers to a single-address instruction.

If the first 3-bit of the instruction is 111, then next 9-bit of the instruction specifies extended opcode (Figure 10(b)).

An instruction without any explicit operand can be encoded with this 9-bit.

Example : CLEAR (clear AC), Skip, Right shift/Left shift, STOP etc.

If the first 3-bit is set to 000, then the next 6 bits specify I/O device address and least significant 3 bits define type of I/O operation (Figure 10(c)).

Data dependent opcode Here tag bits are added with memory words.

In CISC m/c, there are separate integer arithmetic unit and floating point unit.

Such m/c includes multiple instruction for an arithmetic operation. Example - ADD can be defined as ADDinteger, ADDreal and ADDfloating. That is, three ADD instructions are to be included in the instruction set.

In data dependent opcode scheme, these three ADD is replaced by a single addition instruction. Tag of operands specify the type of operands. Example:

ADD r_1, r_2

If r_1 and r_2 are integer, the ADD operation is sent to integer arithmetic unit.

In RISC, use of tag is necessary as RISC considers a very few instructions.

Frequency dependent opcode Some instructions are used frequently in the programs. Example: MOVE, shift (RSHIFT/LSHIFT) etc.

If frequently used instructions are represented in fewer number words, the size of a program can be reduced.

In frequency dependent opcode, the most frequently used instructions are encoded with lesser number of bits. Example: in 8085 microprocessor, MOV R1, R2 is an one byte (one word) instruction. The MOV has 2-bit opcode. The rest 6 bits are to denote two register operands R1 (3-bit), and R2 (3-bit).

MOV	R1,	R2
xx	xxx	xxx

However, opcode of LDA/STA is of 8 bits. These two are 3-byte instructions.

To define frequency dependent opcode, a solution can be Hoffman encoding.

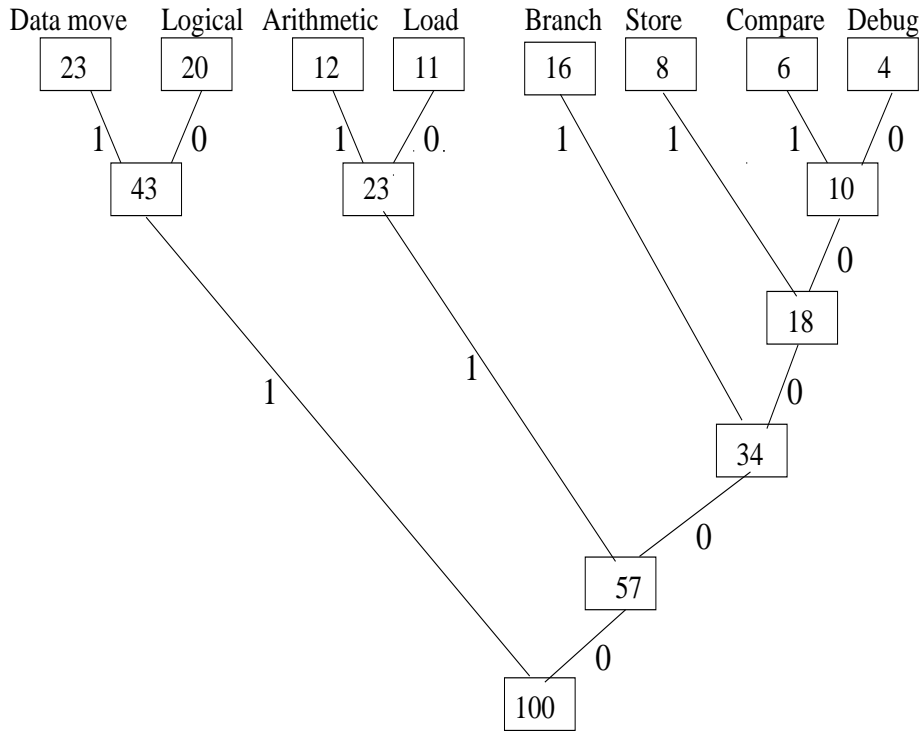


Figure 11: Frequency dependent opcode optimization

Figure 11 describes Hoffman encoding for a m/c with eight type of instructions.

Now, to encode instructions, let assume left child is 1 and right is 0.

A trace from root to leaf defines opcode for the instruction defined by the leaf.

Decoding of variable length opcodes is difficult than that of fixed length opcode.

Number of bits of opcode to be transferred to IR from DR depends on opcode type.

RISC and superscalar computers implement fixed-length instruction format.

0.7 Instruction Implementation

Execution of an (macro) instruction involves execution of a sequence of micro-operations.

0.7.1 Data movement

Here the sources and destinations can be the register or memory.

Implementation of register to register data transfer instruction is shown in Figure 12(a).

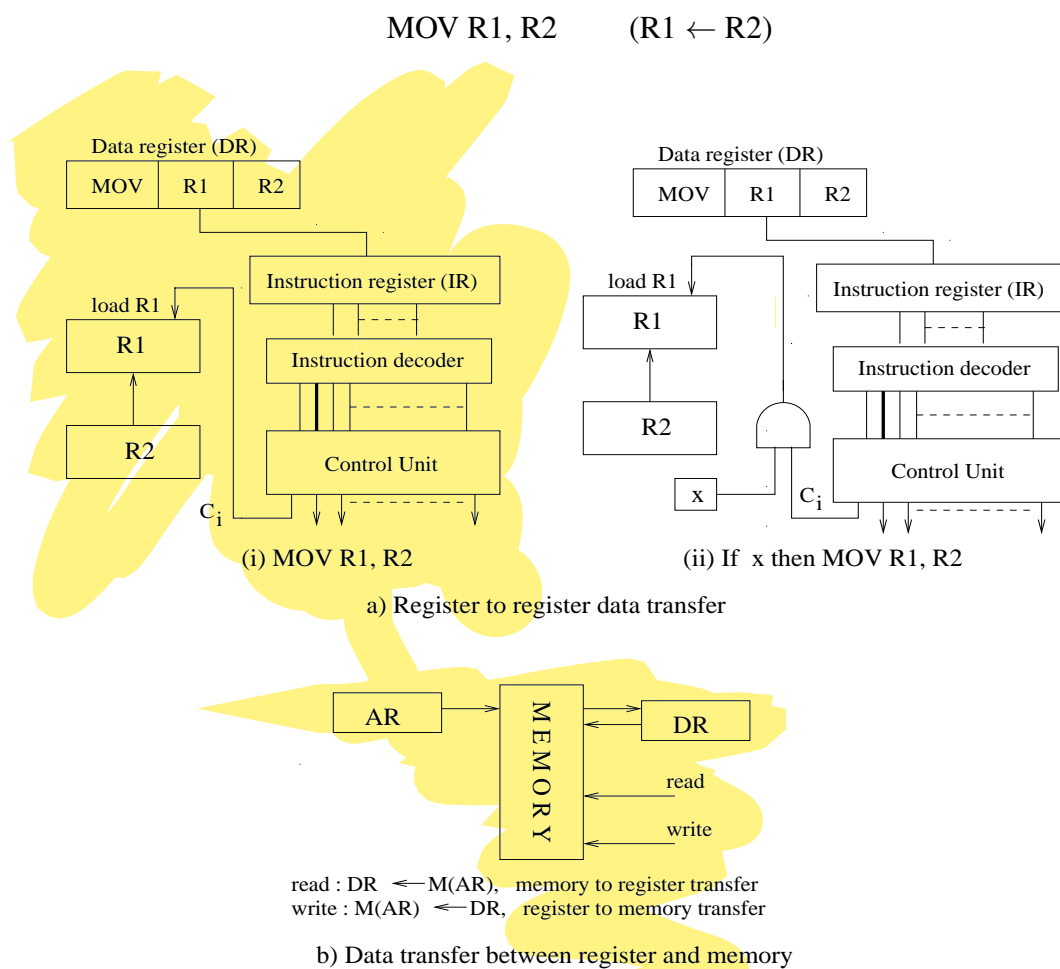


Figure 12: Data transfer instruction implementation

However, the conditional data movement

if x MOV R1, R2 ($R1 \leftarrow R2$)

is realized (as shown in Figure 12(a)(ii)).

The execution of data movement instruction like

MOV A1, A2

where A1/A2 are memory addresses, implies completion of micro-operations

$AR \leftarrow A2$	this is a register to register data transfer
$DR \leftarrow M(AR)$	memory to register transfer
$AR \leftarrow A1$	register to register transfer
$M(AR) \leftarrow DR$	register to memory data transfer

LOAD: memory to register data transfer.

STORE: executes register to memory transfer.

Data movement instruction execution requires bus transfers (Figure 13).

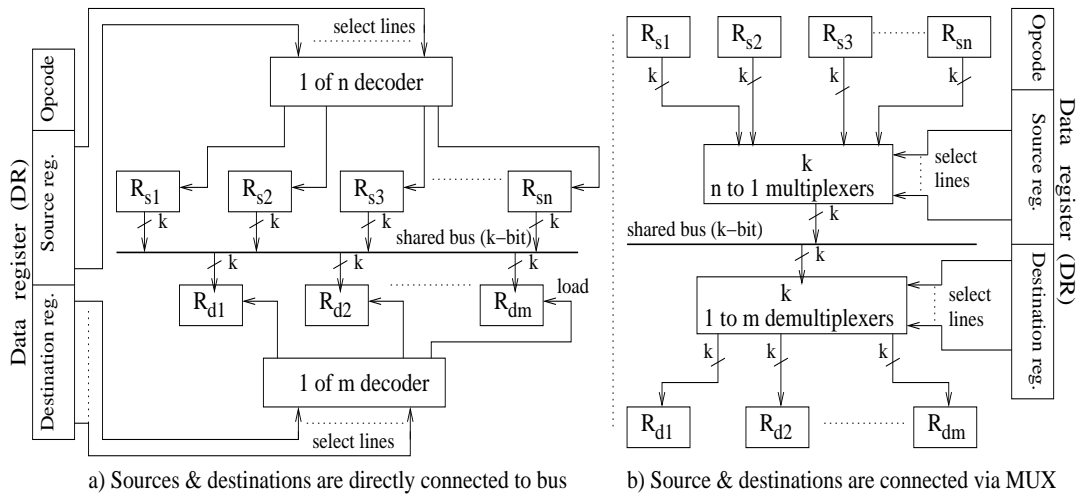


Figure 13: Data transfer from many sources to many destinations

Data movement from R_s to R_d : implies data should be transferred from R_s to bus and then from bus to R_d .

Figure 13 shows data transfer from one of many sources to one of many destinations.

Figure 14: implementation while sources and destinations are same set of registers.

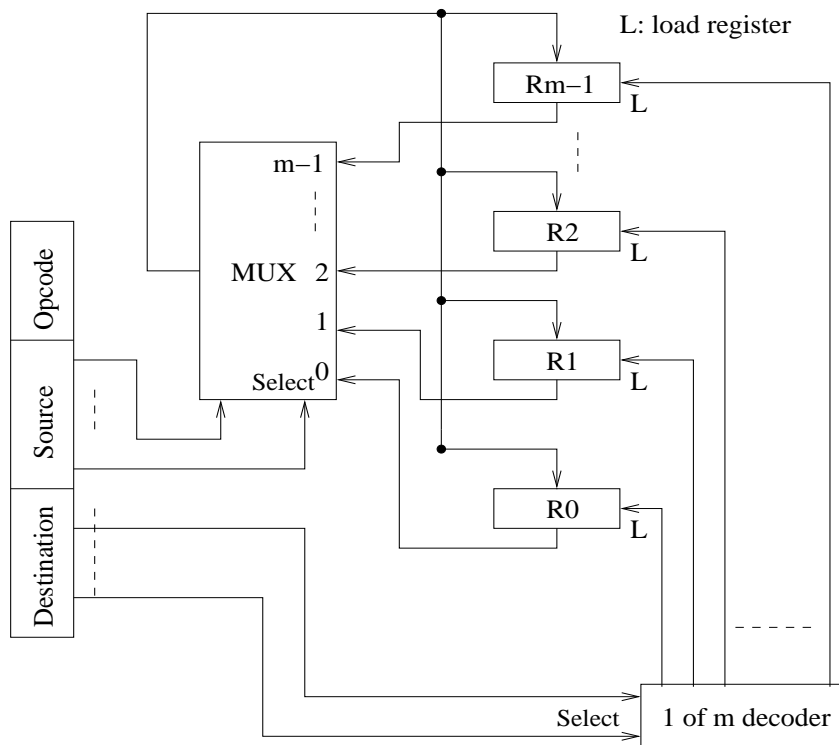


Figure 14: Data transfer among same set of registers

0.7.2 Branch control

The conditional branch, unconditional branch, skip, jump to subroutine etc.

Unconditional branch can be implemented by execution of register (DR) to register (PC) transfer micro-operation (Figure 15) -

$$PC \leftarrow DR \text{ (operand).}$$

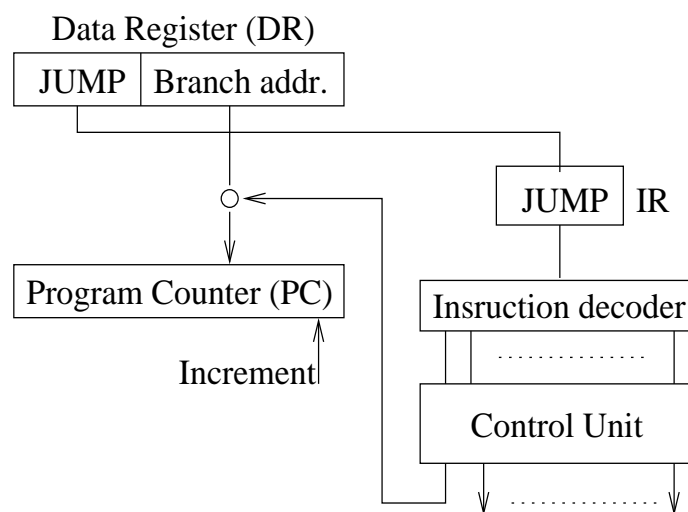


Figure 15: Branch control

Skip implies one instruction is to be skipped.

	⋮
L:	SKIP
L+1:	ADD X, Y, Z
L+2:	I_i
	⋮

ADD instruction will be skipped and next executable instruction will be I_i .

Implementation of unconditional/conditional SKIP is shown in Figure 16.

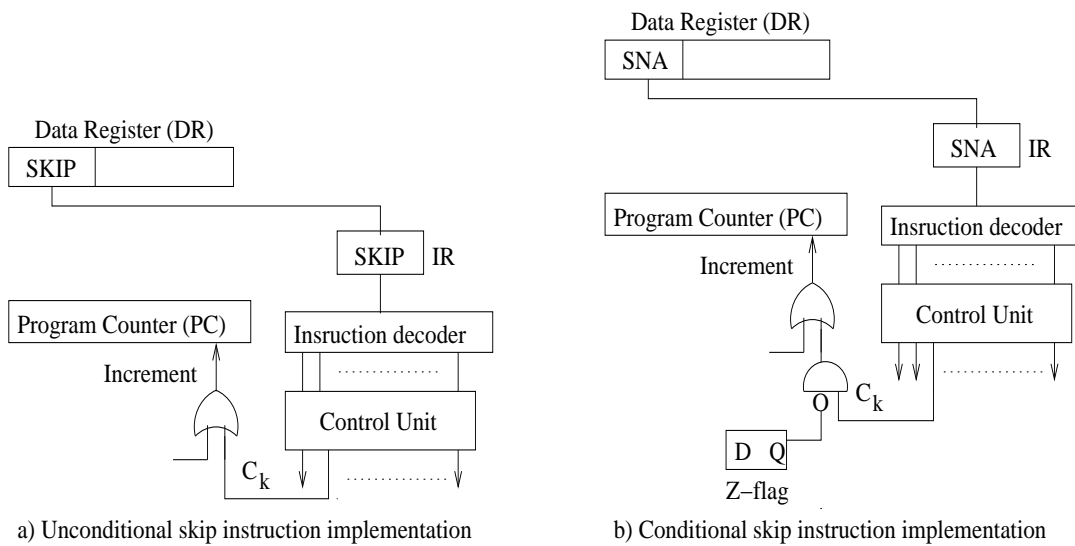


Figure 16: Skip instruction implementation

Conditional branch The conditions (jump on zero/non-zero, jump on positive/negative, jump on parity, jump on carry, jump on even/odd, jump on overflow etc) can be tested from flags (Figure 17).

If condition is satisfied, then register to register to data transfer

$$PC \leftarrow DR (\text{operand}).$$

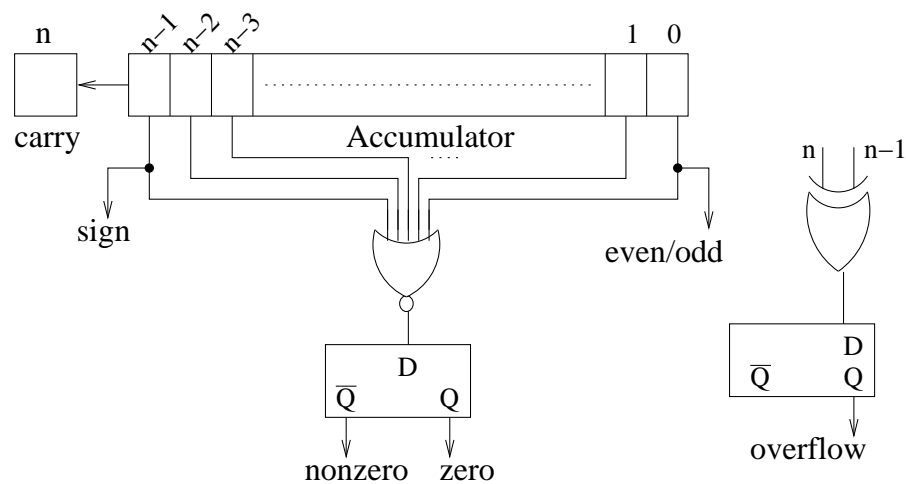


Figure 17: Setting condition flags

Conditional Skip: one instruction is to be skipped if condition is satisfied.

Implementation of SNA (skip on non-zero AC) is shown in Figure 16(b).

Lecture 9: March 2, 2021
Computer Architecture and Organization-I
Biplab K Sikdar

0.7.3 Logical instructions

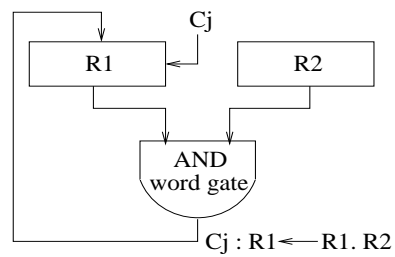


Figure 18: Logical micro-operation implementation

Implementation of logical microoperation is shown in Figure 18.

Figure 19 displays implementation of logical macro instructions of type $Z \leftarrow X \text{ op } Y$, where X, Y, Z are registers. Here, operators (op) are *and/or/xor/not*.

The not operates as $Z \leftarrow X'$.

Control $C1$ and $C2$ of Figure 19(a) is received from instruction decoder, and

$$Z = C1'C2' (X.Y) + C1'C2 (X+Y) + C1C2' (X \oplus Y) + C1C2 X'.$$

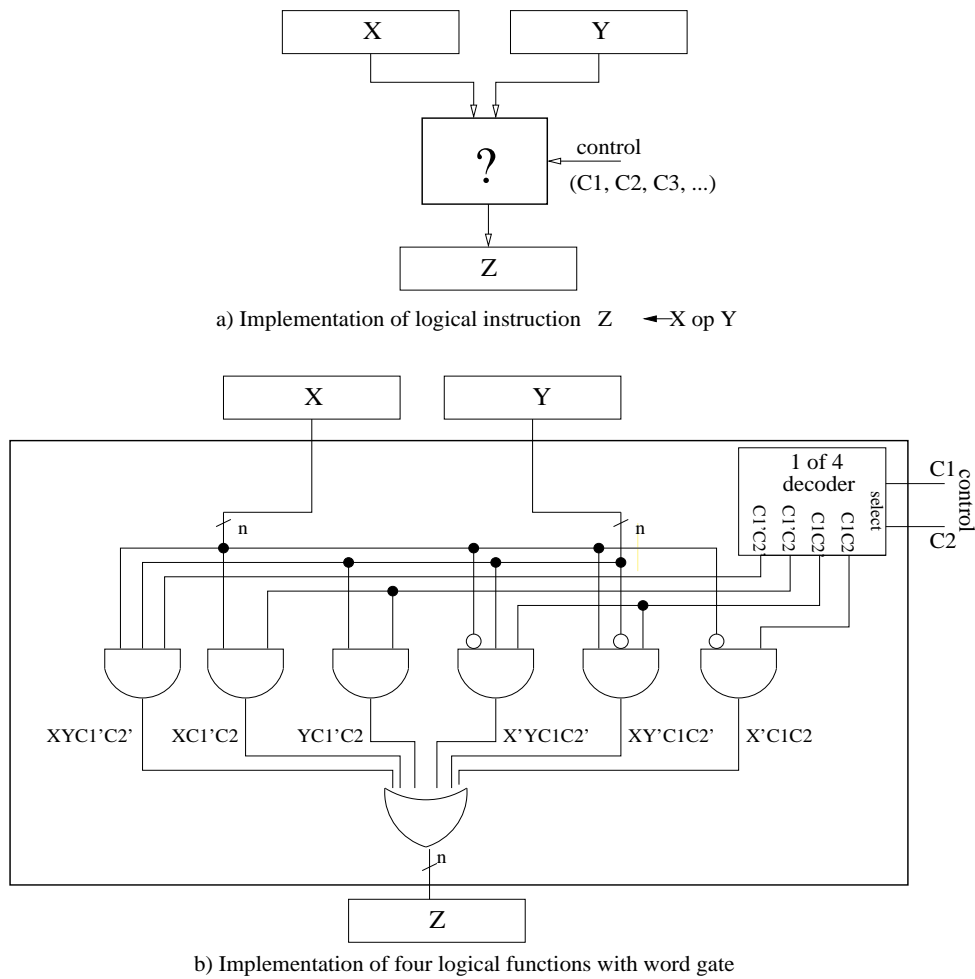


Figure 19: Logical macro-operation implementation

The AND and OR gates of Figure 19(b) are word gates.

0.7.4 Input/output instructions

Input/output instruction basically is a data movement operation (register-to-register transfer).

The very primitive input/output instructions are

IN xx
OUT yy

IN transfers data from input buffer to AC, whereas, OUT transfers data from AC to output buffer (Figure 22).

Activation of control signal C_{in} sets the path from input buffer to AC.

Similarly, C_{out} activates means a path is set from AC to output buffer.

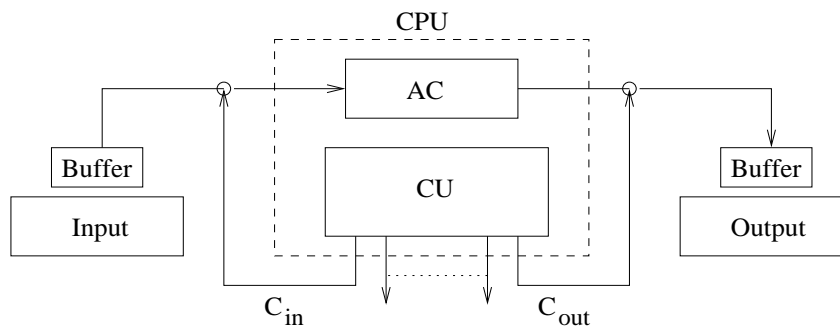


Figure 22: Input-Output instruction implementation

0.8 Arithmetic Instruction Implementations

Arithmetic instruction: a set of data movement and arithmetic micro-operations.

0.8.1 Addition and subtraction

In 2's complement, addition and subtraction micro-operations can be implemented with adder only (Figure 23).

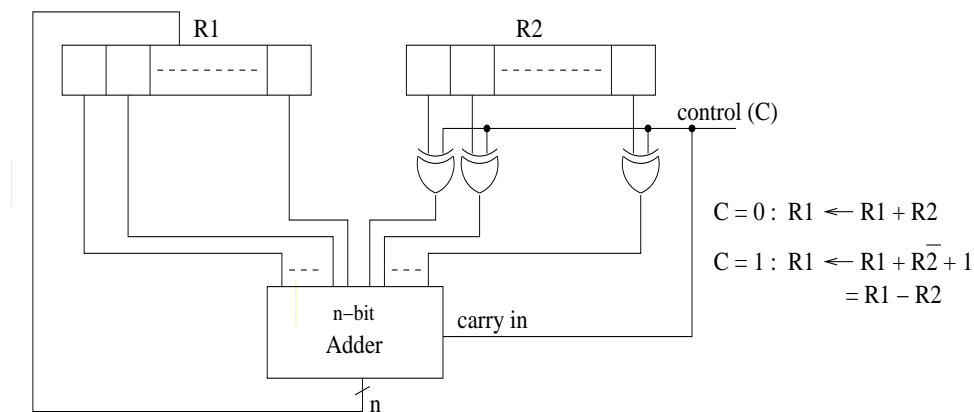


Figure 23: Adder/subtractor

Other arithmetic operations - increment, decrement, ... are implemented with multiple micro-operations and adder/subtractor. For example,

1. Increment R1 implies $R2 \leftarrow 1$ and then $R1 \leftarrow R1 + R2$ ($C=0$),
2. Decrement R1 implies $R2 \leftarrow 1$ and then $R1 \leftarrow R1 - R2$ ($C=1$),

Design adders

Binary adder Truth table of adder is formed and then logic optimization is done with Karnaugh-map (say). A binary adder (Figure 24) computes sum

$$z_i = x_i \oplus y_i \oplus c_{i-1}, \text{ and carry}$$

$$c_i = x_i y_i + x_i c_{i-1} + y_i c_{i-1}$$

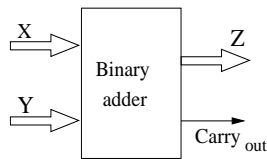


Figure 24: Binary adder

It is the fastest adder. However, number of gates required and the fan-in of each gate grow exponentially with n (number of bits) in the binary adder.

Serial adder Serial adder is shown in Figure 25. If d is the delay of a full adder and delay of F/F is D , then time to add two n -bit numbers is $(d+D)n$.

The hardware requirement in serial adder is independent of n .

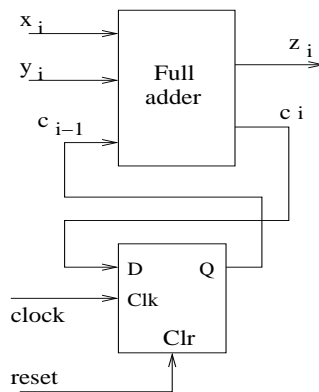


Figure 25: Serial adder

Parallel adder: is also referred to as ripple carry adder.

An n -bit parallel adders requires n full adders (Figure 26).

While adding two n -bit numbers, the carry propagates like ripples (carry generated from the i^{th} full adder (i^{th} stage) is input to the $(i + 1)^{th}$ full adder).

If d denotes the delay of a full adder stage, the worst case addition time $n \times d$.

The hardware grows relative to n .

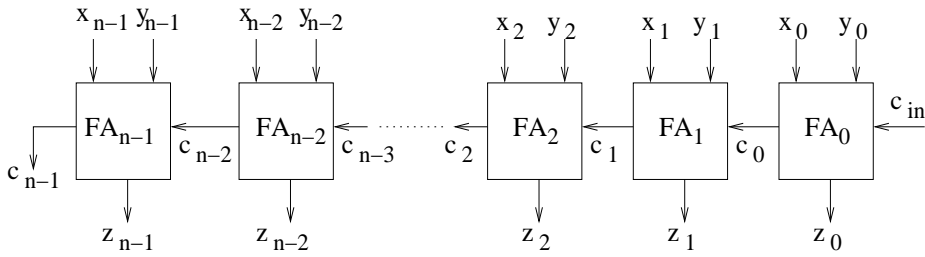


Figure 26: Parallel adder

Carry-Look-Ahead adder Strategy in to reduce the carry propagation delay.

The basic unit of a CLA is shown in Figure 27(a).

The ordinary carry c_i of ordinary full adder is replaced by two signals - carry propagate (p_i) and carry generate (g_i), where

$$p_i = x_i + y_i \text{ and}$$

$$g_i = x_i y_i.$$

The carry to be transmitted to stage $(i+1)$ is

$$c_i = g_i + p_i c_{i-1} \text{ [as } c_i = x_i y_i + c_{i-1}(x_i + y_i)\text{]}.$$

Similarly,

$$c_{i-1} = g_{i-1} + p_{i-1} c_{i-2} \quad \text{-that is, } c_i = g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-2}.$$

By replacing c_{i-2} and so on, c_i can be expressed as SOPs of p s & g s and c_{in} .

Example: Carrys c_0, c_1, c_2 and c_3 are expressed as

$$c_0 = g_0 + p_0 c_{in}$$

$$c_1 = g_1 + p_1 c_0 = g_1 + p_1 g_0 + p_1 p_0 c_{in}$$

$$c_2 = g_2 + p_2 c_1 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_{in}$$

$$c_3 = g_3 + p_3 c_2 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_{in}$$

Figure 27(b) shows the n -bit CLA and the logic circuit for carry c_0 .

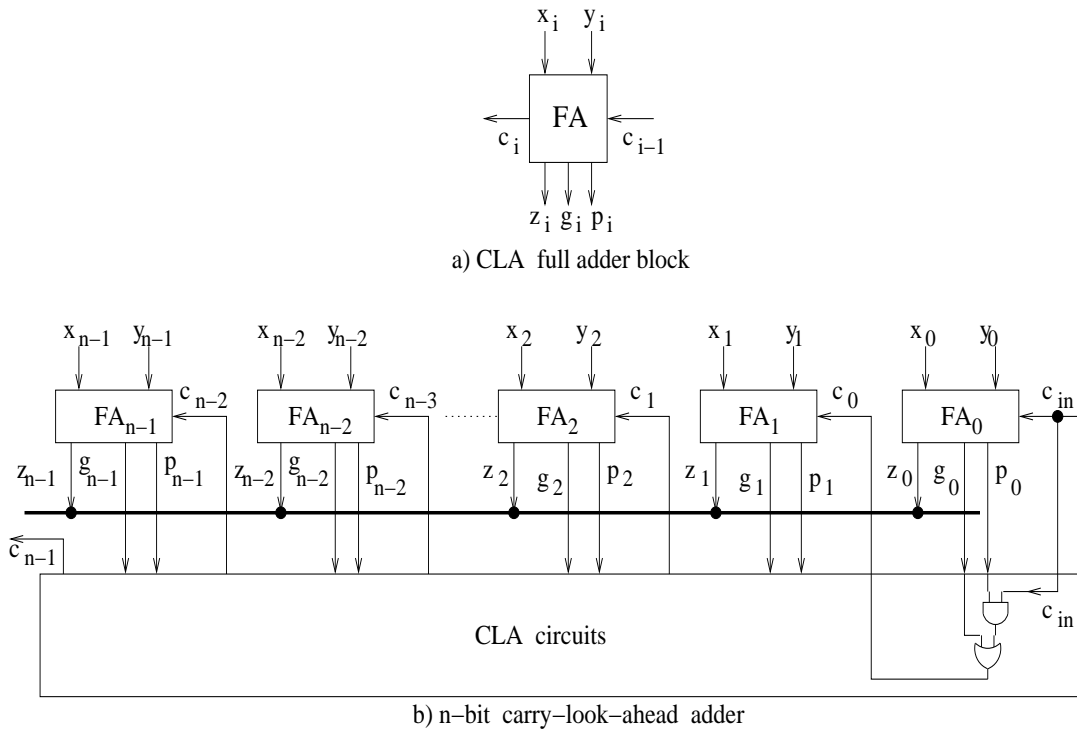


Figure 27: Carry look ahead adder

If d is the propagation delay of FA/..., the time required to compute sum of two n -bit numbers is $(d+d+d) = 3d$ -that is, independent of n .

In first d , all the p_i s and g_i s are generated. During second d carries (c_i s) are calculated. In third d , z s (SUM) are computed.

Design complexity of a CLA adder increases with the value of n .

Fan-in/fan-out of AND/OR gates for computing carry become unrealistic.

This restricts the design of CLA for 16/32/64-bit adders.

In practice, 4-bit to 8-bit CLA adders are designed.

16-bit/32-bit/64-bit adder is implemented with cascade of 4/8-bit CLA adders.

16-bit adder using four 4-bit CLA adders:

16-bit adder of Figure 28 is designed with four 4-bit CLAs.

The 4-bit CLA modules are designed as in Figure 27(b).

Final carry generated from i^{th} CLA module is input to $(i + 1)^{th}$ CLA module.

Time required to compute two 16-bit numbers in such an adder is

$$3d + d + d + d = 6d.$$

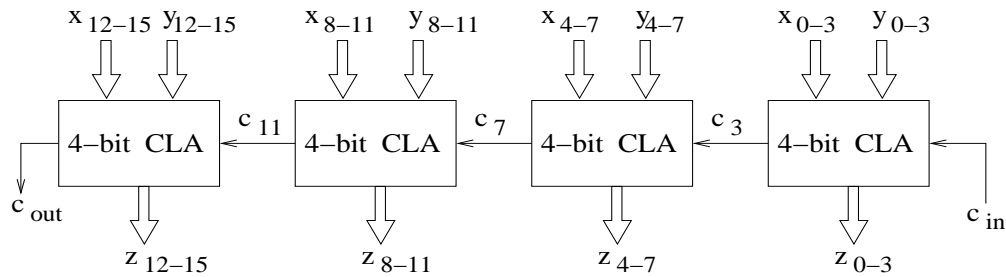


Figure 28: Cascaded carry look ahead adder

Lecture 10: March 3, 2021
Computer Architecture and Organization-I
Biplab K Sikdar

Carry Save Adder (CSA)

Carry save adder (CSA) is effective while adding more than two numbers.

Example: addition of three n -bit numbers (in CSA)

An n -bit CSA consists of n -disjoint full adders (Figure 29(b)).

Addition four n -bit numbers is shown in Figure 29(a).

Addition of six n -bit numbers is shown in Figure 29(c).

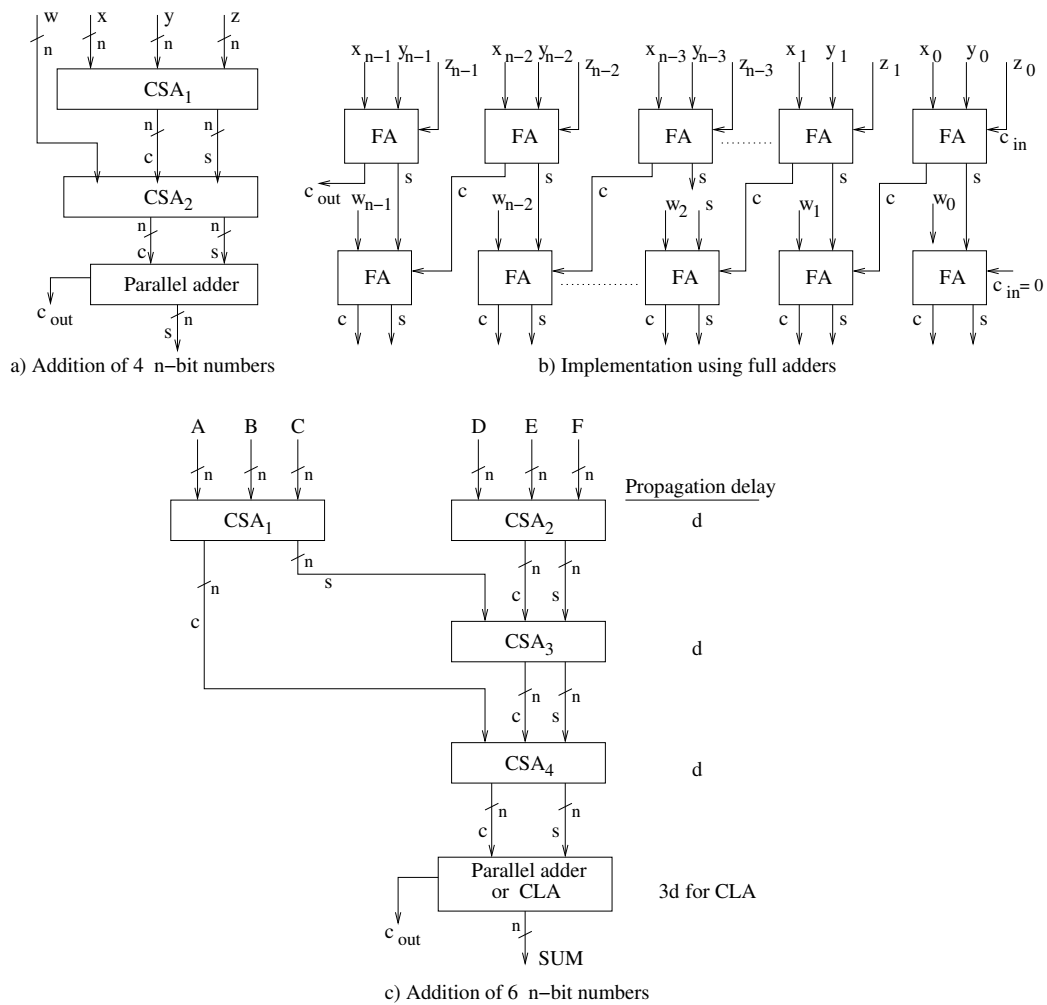


Figure 29: Carry save addition

0.9 Multiplication Instruction Implementation

0.9.1 Multiplication in sign magnitude

Simplest implementation of fixed point multiplication instruction is using counters.

Let multiplicand is P and Q is the multiplier.

Product is targeted to store in CP.

In counter based implementation, CP is a counter.

Following steps computes $CP = P \times Q$. all four QC, CQ, MC and CP are counters.

1. Let $QC \leftarrow multiplier$; $CQ \leftarrow multiplier$; $MC \leftarrow multiplicand$; $CP \leftarrow 0$
2. If MC and/or CQ = 0, then exit
3. Decrement CQ [$CQ = CQ - 1$]; Increment CP [$CP = CP + 1$]
4. If $CQ \neq 0$, then go to Step 3
5. Decrement MC [$MC = MC - 1$]; Copy QC to CQ [$CQ \leftarrow QC$]
6. If $MC \neq 0$, then go to Step 3
7. Output CP as product

This method is simple but very slow.

Alternative implementation can be add multiplicand (M) Q (multiplier) times.

That is,

Initialize PRODUCT ($M \times Q$) = 0 and then

Perform PRODUCT = PRODUCT + M \cdots Q times.

This implementation requires a counter to store the multiplier.

Multiplication of n -bit numbers in sign magnitude can also be implemented following the steps used in multiplication of decimal numbers (shift/addition technique).

Example: Multiplication of 8-bit numbers

$$Y = y_7y_6 \cdots y_1y_0 = 01100101 \text{ and } X = x_7x_6 \cdots x_1x_0 = 11011101.$$

To compute magnitude of product $P = p_6p_5 \cdots p_1p_0$, the 7 magnitude bits of Y and X are to be multiplied.

	1100101	$y_6y_5y_4y_3y_2y_1y_0$
	1011101	$x_6x_5x_4x_3x_2x_1x_0$
_____	_____	
0000000	1100101	P_0
0000000	0000000	P_1
0000011	0010100	P_2
0000110	0101000	P_3
0001100	1010000	P_4
0000000	0000000	P_5
0110010	1000000	P_6
_____	_____	

P_i is the partial product. Therefore, the magnitude of product is

$$P = \sum_{i=0}^{n-2} 2^i Y x_i$$

Sign of P -that is, $p_7 = 1$ (XOR of y_7 and x_7).

Lecture 11-12: March 5, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

Multiplication of n -bit numbers in sign magnitude shift/addition technique

$$Y = y_7y_6 \cdots y_1y_0 = 01100101 \text{ and } X = x_7x_6 \cdots x_1x_0 = 11011101.$$

To get product $P = p_6p_5 \cdots p_1p_0$, 7 magnitude bits of Y and X are to be multiplied.

Here, P is the sum of partial products P_0, P_1, \dots, P_6 .

A partial product can be all 0s or $Y \times 2^i$ (that is, Y with i left shift).

	1100101	$y_6y_5y_4y_3y_2y_1y_0$
	1011101	$x_6x_5x_4x_3x_2x_1x_0$
_____	_____	
0000000	1100101	P_0
0000000	0000000	P_1
0000011	0010100	P_2
0000110	0101000	P_3
0001100	1010000	P_4
0000000	0000000	P_5
0110010	1000000	P_6
_____	_____	

In m/c, we can implement it by right shift of the register that stores product P.

Consider accumulator A and register Q stores the product.

A stores most significant part of P, and Q stores least significant part of P.

A and Q are connected and form a single shift register.

Algorithm 0.1 In sign magnitude, we need to consider multiplier[$n-2:0$], multiplicand[$n-2:0$] and A[$n-2:0$].

Step 1: $A[n-2:0] \leftarrow 0$; $C \leftarrow 0$; $M[n-2:0] \leftarrow \text{multiplicand}[n-2:0]$; $Q[n-2:0] \leftarrow \text{multiplier}[n-2:0]$

Step 2: if $Q_0 \neq 0$, then $A \leftarrow A+M$

Step 3: right shift A, Q

Step 4: if $C \neq n-2$ then increment $C = C+1$ and go to Step 2.

Step 5: output A, Q and exit.

Example 0.2 Here $n=5$. The magnitude is of 4-bit.

Action	CY	A	Q	M	C
	0	0000	1101	1010	0
+		1010			
	0	1010			
RS	0	0101	0110		1
RS	0	0010	1011		2
+		1010			
	0	1100			
RS	0	0110	0101		3
+		1010			
	1	0000			
RS	1	1000	0010		

$$10 \times 13 = 130$$

0.9.2 Fixed point multiplication in 2's complement

Multiplication in 2's complement can be done by modifying Algorithm 0.1.

Here, sign bits of multiplier and multiplicand are treated as magnitude.

Consider $M = 0110 = 6$ and $Q = 1101 = -3 = 13 - 16$ in 2's complement.

If Algorithm 0.1 is followed, product $P_{sign} = 0100\ 1110 = 78$. But it should be -18.

The final product P needs correction as

$$P = P_{sign} - 2^4 \times M = 78 - 16 \times 6 = -18.$$

Let multiplicand $Y = y_{n-1} \cdots y_1 y_0$ and multiplier $X = x_{n-1} \cdots x_1 x_0$ in 2's complement.

- Case I: $y_{n-1} = x_{n-1} = 0$: Algorithm 0.1 can produce the correct result.
- Case II: $y_{n-1} = 1$ and $x_{n-1} = 0$: ($Y = 1101 = -3$ and $X = 0110 = +6$).

As per Algorithm 0.1 product P_{sign} is 78 (incorrect).

Reason is - Algorithm 0.1 assumes partial products ($P_0 = 0000\ 0000$, $P_1 = 0001\ 1010$, $P_2 = 0011\ 0100$, and $P_3 = 0000\ 0000$) $P_i = 2^i Y x_i \forall_{i=0}^{n-1}$ as positive.

As multiplicand is negative, all partial products must be negative.

Correction needed in Step 3 of Algorithm 0.1: enter 1 in MSB if partial product is not 0.

- Case III: $y_{n-1} = 0$ and $x_{n-1} = 1$: Algorithm 0.1 results in incorrect product $P_{sign} = (2^n + X)Y$.

The correction needed is

$$P = P_{sign} - 2^n Y.$$

- Case IV: $y_{n-1} = x_{n-1} = 1$: Algorithm 0.1 has to be corrected as in Case II/III.

Algorithm 0.2 Step 1: $A[n-1:0] \leftarrow 0$; $CY \leftarrow 0$; $Count \leftarrow 0$;
 $M[n-1:0] \leftarrow \text{multiplicand}[n-1:0]$; $Q[n-1:0] \leftarrow \text{multiplier}[n-1:0]$
Step 2: if $Q_0 \neq 0$, then $A \leftarrow A+M$
Step 3: right shift A, Q
 $AQ_i \leftarrow AQ_{i+1} \forall_{i=0}^{n-2}$
 $A_{n-1} \leftarrow A_{n-1} \vee CY$
Step 4: if $Count \neq n-2$ then increment $Count = Count+1$; go to Step 2.
Step 5: if $Q_0 = 0$, then right shift; go to Step 8
Step 6: $A \leftarrow A+M$; right shift
Step 7: $A \leftarrow A-M$
Step 8: Output A,Q

Action	CY	A	Q	M	Count
	0	0000	1101 <u>1</u>	1010	0
+		1010			
	0	1010			
RS	0	1101	0110 <u>1</u>		1
RS	0	1110	101 <u>1</u>		2
+		1010			
	1	1000			
RS	1	1100	010 <u>1</u>		
+		1010			
	1	0110			
RS	1	1011	0010 <u>1</u>		
—		1010			
		0001	0010	Product	

This requires at least p additions/subtractions (p : number of 1s in multiplier). Further, correction (subtraction) is needed if multiplier is negative.

0.9.3 Booth's algorithm for multiplication in 2's complement

Andrew Donald Booth (British) has proposed Booth's algorithm.

Let consider computation of Product $P = M \times Q$, where Q is the multiplier and

$$Q = \begin{matrix} & i=4 & & 3 & 2 & 1 & 0 \\ & 0 & & 1 & 1 & 1 & 0 \end{matrix} = 14.$$

Therefore, $P = 14 \times M = (+16 - 2) \times M = +2^4 \times M - 2^1 \times M$.

Similarly, for the multiplier

$$Q = \begin{matrix} & i=11 & & 10 & 9 & & 8 & 7 & 6 & & 5 & 4 & 3 & & 2 & 1 & 0 \\ & 0 & & 0 & 1 & & 1 & 1 & 0 & & 0 & 1 & 1 & & 1 & 1 & 0 \end{matrix},$$

$$P = +2^{10} \times M - 2^7 \times M + 2^5 \times M - 2^1 \times M.$$

Total number of additions/subtractions is 4 only. It signifies

- (i) Number of additions/subtractions may be less than the number of 1s in multiplier. It depends on number of flips (0 to 1 or 1 to 0) in multiplier.
- (ii) If i^{th} and $(i-1)^{th}$ bit-pair of multiplier is 10, then subtraction is needed.
- (iii) If i^{th} and $(i-1)^{th}$ bit-pair of multiplier is 01, then addition is required.

Features In Booth's algorithm of 2's complement multiplication,

- i) No correction is needed as required for Algorithm 0.2,
- ii) Negative and positive numbers are treated uniformly,
- iii) Computation of product is faster than Algorithm 0.2.

While computing product of Booth's algorithm scans multiplier $X = x_{n-1}x_{n-2} \cdots x_2x_1x_0$ considering adjacent bits $x_i x_{i-1}$.

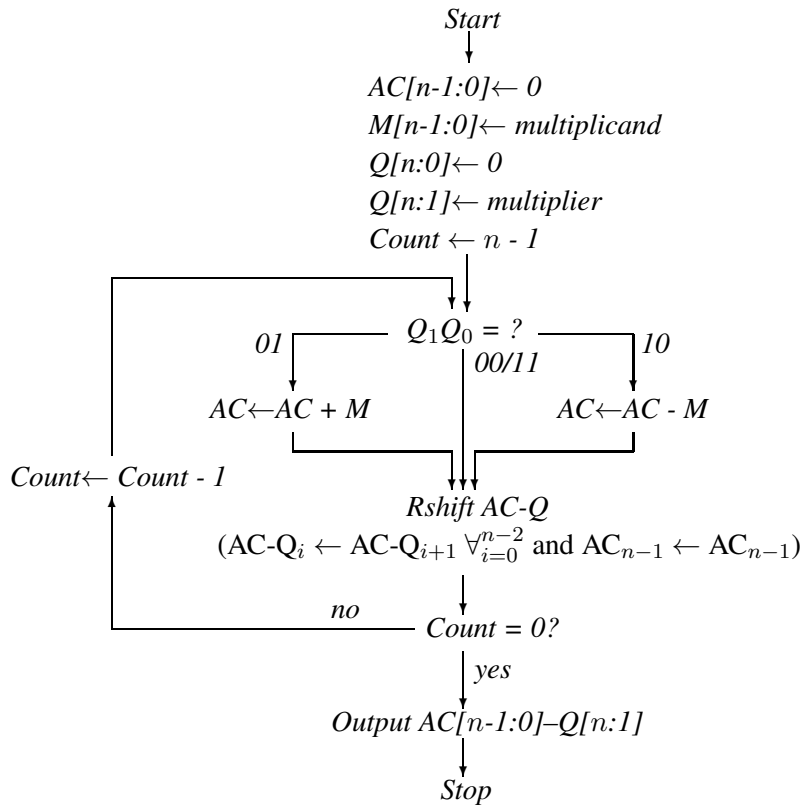
- If $x_i x_{i-1} = 01$, multiplicand Y is added to partial product PP_{i-1} .
- If $x_i x_{i-1} = 10$, Y is subtracted from PP_{i-1} .

Followed by a left shift to get PP_i . Here, PP_i defines product of Y and $x_i x_{i-1} \cdots x_2 x_1 x_0$.

- If $x_i = x_{i-1}$, then only left shift of PP_{i-1} is carried out to get PP_i .

LSB of multiplier X is the $i=0^{th}$ bit.

A bit $x[0]=0$ is appended with LSB of X to facilitate bit pair for $i=0$.



Multiplicand M = 1010 and multiplier Q is 1101.

Action	AC	Q	M	Count
	0000	110 <u>1</u> 0	1010	3
—	1010			
	0110			
RS	0011	011 <u>0</u> 1		2
+	1010			
	1101			
RS	1110	101 <u>1</u> 0		1
—	1010			
	0100			
RS	0010	010 <u>1</u> 1		0
RS	<u>0001</u>	<u>0010</u> 1		

The product is $AC[3:0]-Q[4:1] = 00010010$.

Hardware realization of Booth's multiplier is in Figure 30.

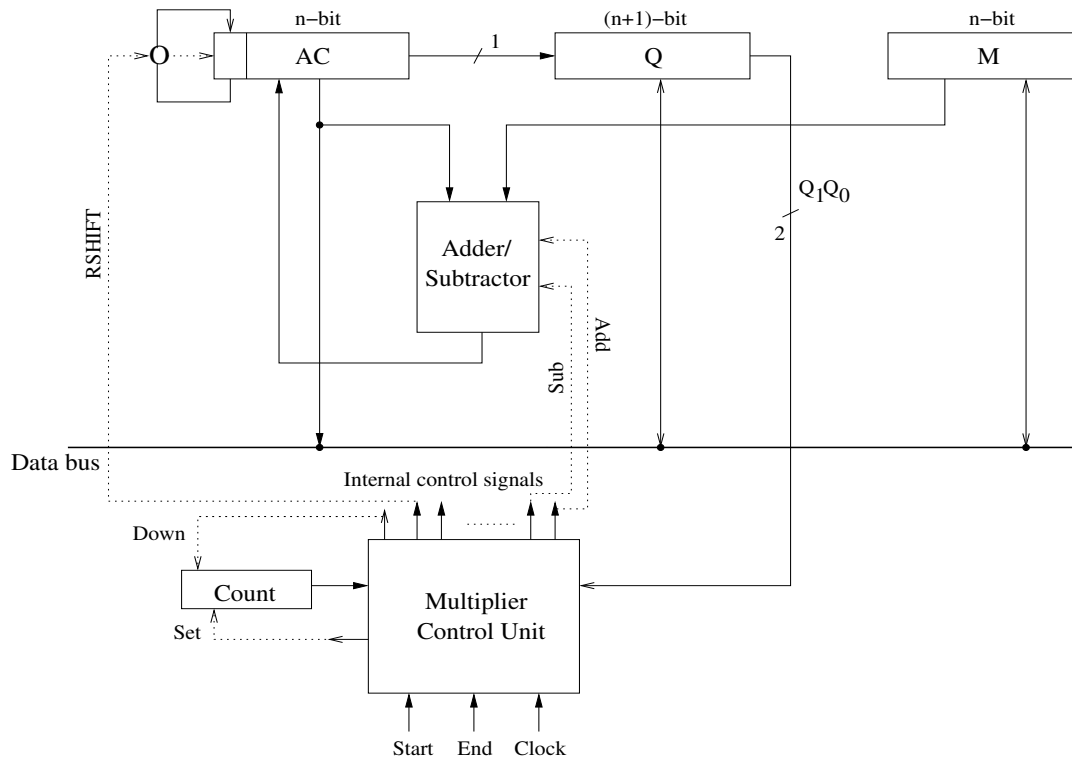


Figure 30: Booth's multiplication algorithm hardware realization

Multiplier control unit compares two least significant bits of register Q and generates appropriate control signals (Add, Sub, RSHIFT, Down Counter, etc).

Limitations: Booth's algorithm enhances speed of multiplication for runs of 1s in multiplier.

For an n -bit multiplier with $\frac{n}{2}$ isolated 1s, Booth's algorithm becomes more costly.

For such a case, Booth's algorithm needs n additions/subtractions.

Let multiplier $X = 01010101$. Number of additions/subtractions needed 8.

$$X - \text{Multiplier} = 01010101$$

$$Q - \text{Multiplier with augmented 0 at least significant position} = 010101010$$

$$\text{Number of 01 pairs in } Q \text{ (additions in Booth's algorithm)} = 4$$

$$\text{Number of 10 pairs (subtractions in Booth's algorithm)} = 4$$

0.9.4 Bit-pair multiplication scheme

If $(i+1)^{th}$, i^{th} , and $(i-1)^{th}$ bits in multiplier are 101, then as per Booth's algorithm,

$$\begin{aligned} PP_{i+1} &= PP_{i-1} + 2^i Y - 2^{i+1} Y \\ &= PP_{i-1} - 2^i Y. \end{aligned}$$

So, for bit pair 101, a subtraction can replace an addition and a subtraction.

Similarly, for 010

$$\begin{aligned} PP_{i+1} &= PP_{i-1} - 2^i Y + 2^{i+1} Y \\ &= PP_{i-1} + 2^i Y. \end{aligned}$$

Hence two arithmetic operations can be replaced by a single operation.

Actions to be taken for all such 8 bit pairs are described in following table.

$i+1$	i	$i-1$	Action
0	0	0	$PP_{i+1} = PP_{i-1}$
0	0	1	$PP_{i+1} = PP_{i-1} + 2^i Y$
0	1	0	$PP_{i+1} = PP_{i-1} + 2^i Y$
0	1	1	$PP_{i+1} = PP_{i-1} + 2^{i+1} Y$
1	0	0	$PP_{i+1} = PP_{i-1} - 2^{i+1} Y$
1	0	1	$PP_{i+1} = PP_{i-1} - 2^i Y$
1	1	0	$PP_{i+1} = PP_{i-1} - 2^i Y$
1	1	1	$PP_{i+1} = PP_{i-1}$

Lecture 13-14: March 12, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

Control Unit

Other than the ALU the important block of a CPU is control unit (CU).

CU interprets CPU instruction to determine control signals to be issued for instruction execution. Input outputs of a CU are shown in Figure 1.

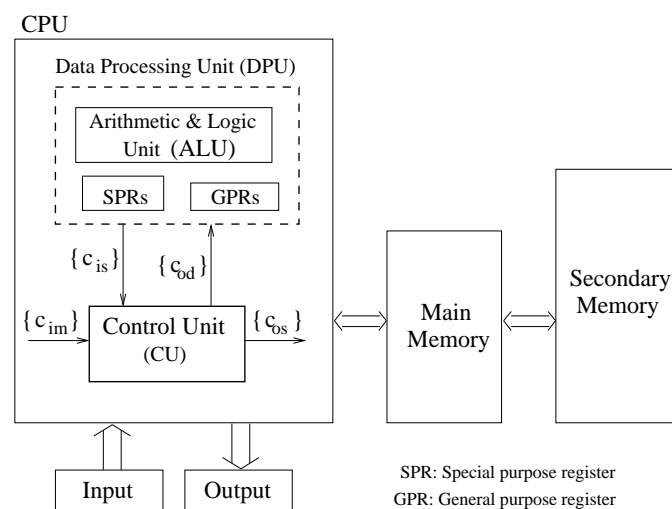
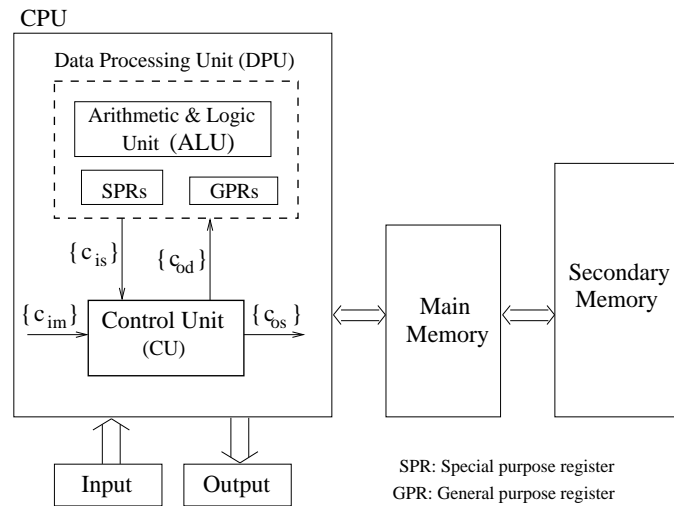


Figure 1: Input output signals of a control unit

- c_{od} s directly control operations of DPU (data processing unit). Main function of a CU is to generate c_{od} s.
- c_{is} s are status signals of CU. These allow CU to make data dependent decision, such as errors, overflow in DPU etc.
- c_{os} is transmitted to other CUs and indicates status conditions such as 'busy' or 'operation completed'.
- c_{im} s are received from other CUs that is, signals from a supervisor.

0.1 Control Design



The main functions of a control unit for a CPU are to

- i) Fetch an instruction from memory -that is, instruction sequencing,
- ii) Interpret instruction in determining control signals to be sent to DPUs -that is, instruction interpretation.

0.1.1 Instruction sequencing

For instruction sequencing, the simplest method can be the storing of next executable instruction address with in the current instruction (Figure 2).

Opcode	Operands	Next Instruction Address
--------	----------	--------------------------

Figure 2: Instruction format containing next instruction address

This technique was followed in early designs (example EDVAC).

Inclusion of next instruction address in instruction format increases instruction length.

Alternative scheme: use of PC or IAR (instruction address register).

PC stores next executable instruction address.

While CPU executing an instruction I_i , PC is incremented by 1 (or k) to point to next executable instruction I_j .

$$PC \leftarrow PC + 1$$

or

$$PC \leftarrow PC + k$$

where k memory word is required to store the current instruction I_i in execution.

For a branch instruction, say JMP X, PC will be loaded with X -that is,

$$PC \leftarrow X \text{ for JMP X}$$

For conditional branch, it is

$$PC \leftarrow X$$

if branch condition is true. Otherwise,

$$PC \leftarrow PC + 1(k).$$

Instruction sequencing in subroutine call (CALL X (and RET)) is implemented as

$$\text{CALL X} \equiv \text{PUSH PC}, \quad \text{RET} \equiv \text{POP PC}.$$

0.1.2 The Design

Two basic techniques are used to design a control unit for the CPU.

A. Hardwired control: Follows design of a sequential logic circuit to generate specific fixed sequence of control signals. Once constructed, changes can only be implemented by redesigning and physically viewing the unit.

B. Microprogrammed control: Execute sequence of micro-instructions for a particular macro-instruction (machine assembly instruction).

A micro-instruction is a set of micro-operations.

A micro-operation performs a data transfer between registers, data transfer between a register and a bus, or simple arithmetic/logical operation.

Execution of a macro-instruction is performed by fetching micro-instructions one at a time from specially designed control memory (CM).

Decoding of micro-instruction enables activation of control lines (signals).

Control signals are implemented in software rather than the hardware and, therefore, design changes are easy.

For design changes, we need to alter content of control memory.

Micro-programmed control unit is slower than hardware design.

0.2 Hardwired Control Design

Design methodologies considered for hardware design of CU - (i) State-table method, (ii) Delay element method, and (iii) Sequence counter method.

0.2.1 State-table method

CU is a finite state machine (FSM). Design steps follow logic of designing an FSM.

This technique is suitable for small CU. There are several practical disadvantages -

(a) Number of states & input condition may be very large and thus state table size & amount of computation needed become excessive;

(b) Very complex circuit design.

Design debugging and subsequent maintenance of circuit become more difficult.

0.2.2 Delay element method

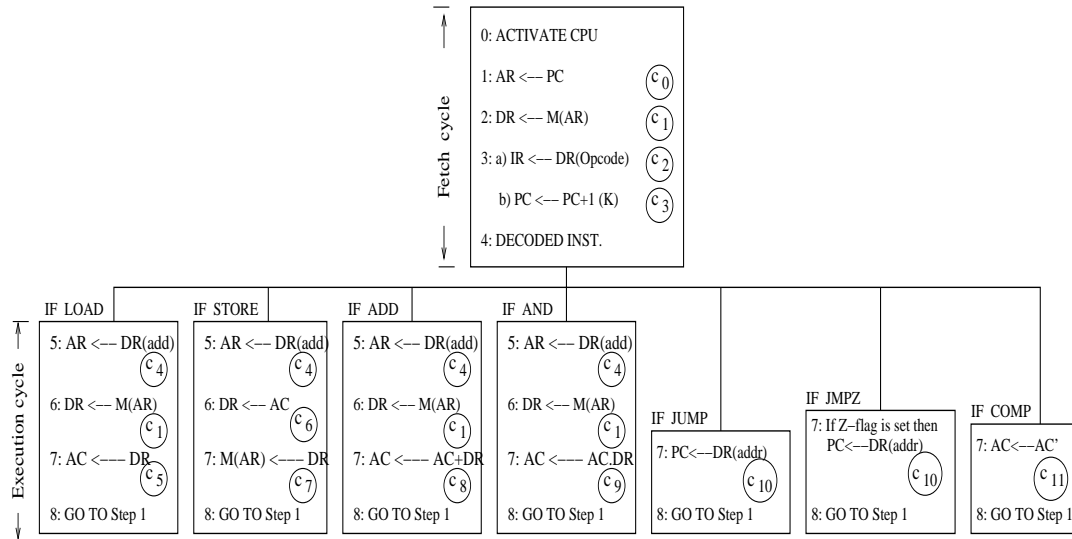


Figure 3: The sequence of control signals

Consider the function of CU shown in Figure 3, of a CPU with 7 macro-instructions.

A simplest version showing only sequence of control signals is in Figure 4.

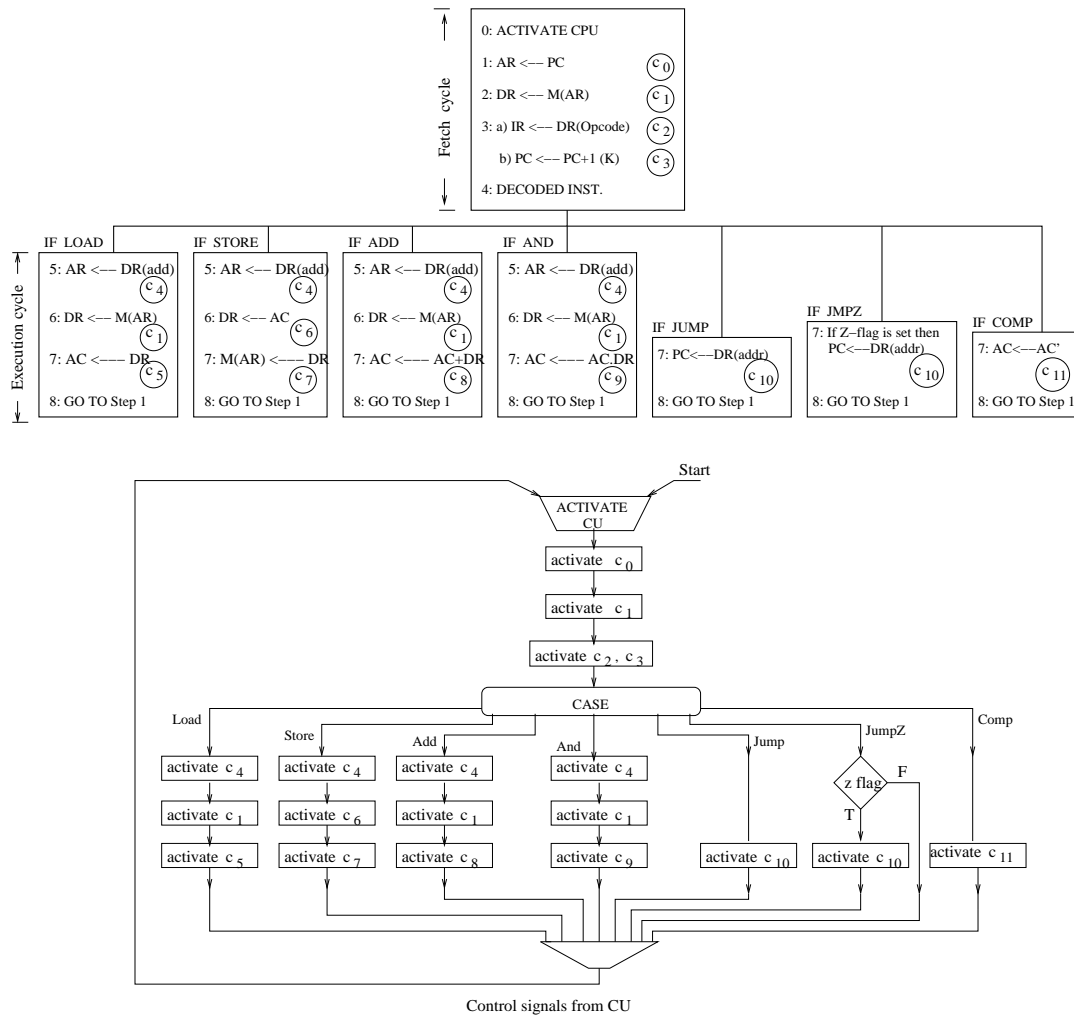
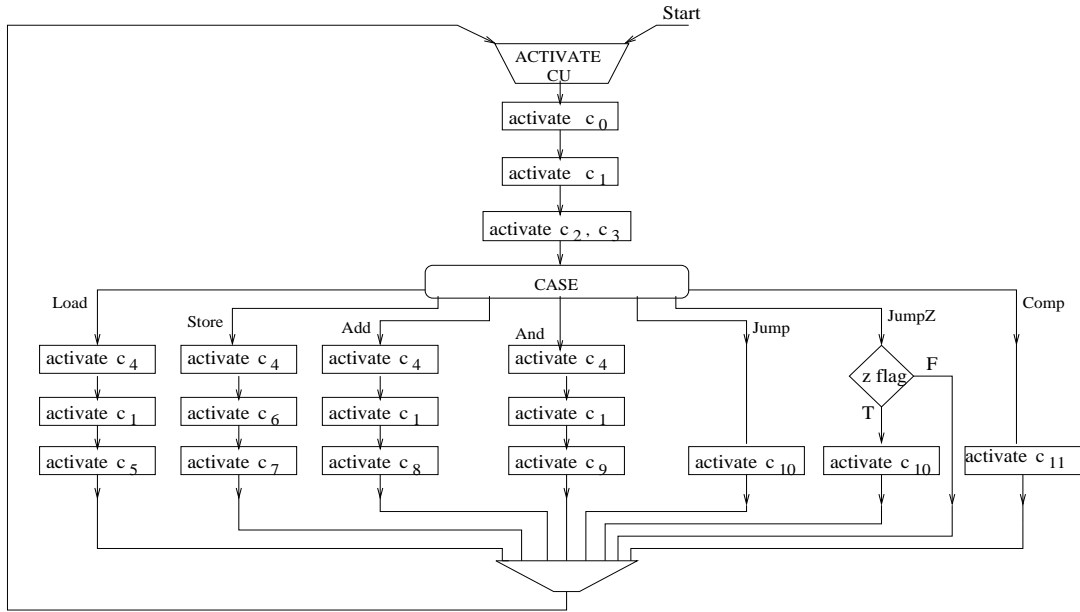


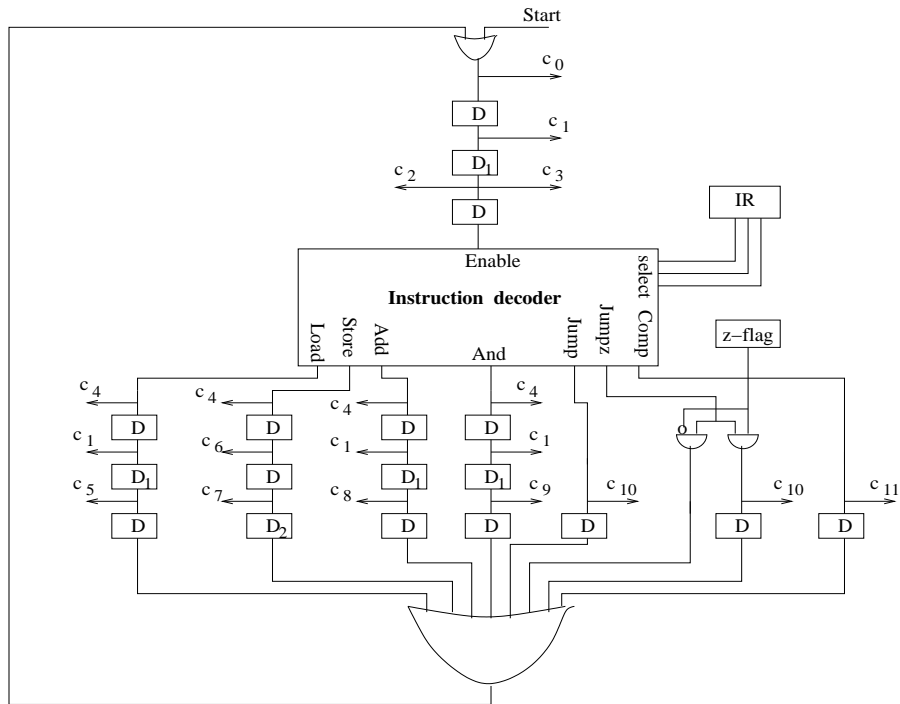
Figure 4: Sequence of control signals

CU designed with delay elements is shown in Figure 5.

CU using delay elements is designed directly from the diagram.



(a) Control signals from CU



(b) CU designed in delay element method

Figure 5: Example design with delay element method

Following rules are followed for such a design:

- a) In between two successive control signals (micro-instructions), a delay element is to be inserted.

That is, an *assignment box* in control flow diagram is replaced by a edge and an *edge* is replaced by a delay element (Figure 6(a)).

- b) The *k to 1 connector* is replaced by a *k*-input OR gate (Figure 6(b)).

- c) *Decision box* is replaced by two 2-input AND gates (Figure 6(c)).

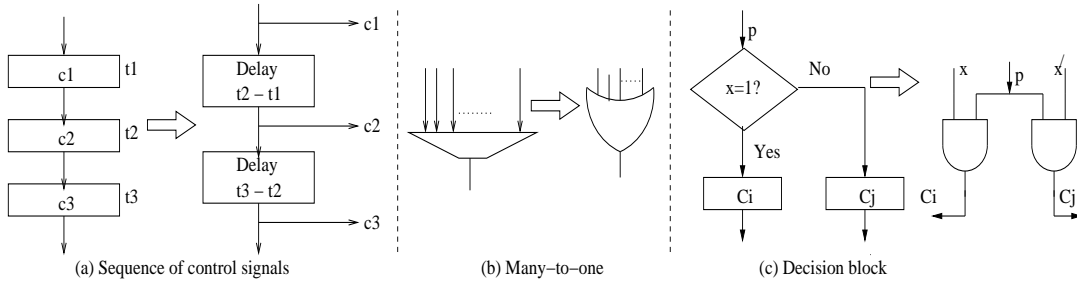


Figure 6: Control flow to logic conversion

Output of a delay element is to be a signal pulse of precise magnitude and duration.

In real design requirement, duration of different control signals may not be the same.

In example design of Figure 5(b), we assume all the control signals except c_1 and c_7 are to be of same duration (D).

Both c_1 and c_7 manage memory access (read/write) and, therefore, are to be activated for larger period of time.

In example, delays are shown as D_1 and D_2 respectively.

Limitations

Main limitation of control design following delay element method is the number of delay elements needed (approximately equal to the number of states $O(N)$).

Synchronization of so many widely distributed delay elements is difficult.

Design of a huge number of delay elements with specific delays is very expensive.

0.2.3 Sequence counter method

Design of CU using sequence counter method is based on the fact that control circuits perform a relatively small number of tasks repeatedly.

CU of Figure 5 repeats 6-tasks (that is, activate c_0 , c_1 , (c_2 , c_3), c_4 , c_1 , c_5 ; if CPU executes a series of Load instructions).

A better representation of tasks performed repeatedly is shown in Figure 7.

Each pass in loop of Figure 7 constitutes an CPU instruction cycle.

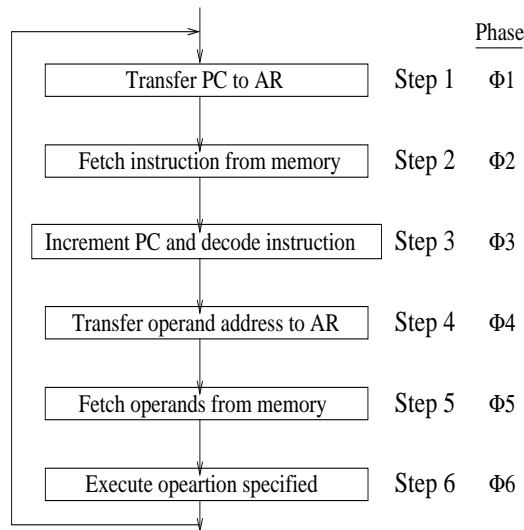


Figure 7: CPU behaviour represented as a single closed loop

If each step in loop is performed in an appropriately chosen clock period (ϕ), a CU can be built around a single modulo-6 sequence counter.

It is assumed that each step of Figure 7 including memory access is performed in one clock period.

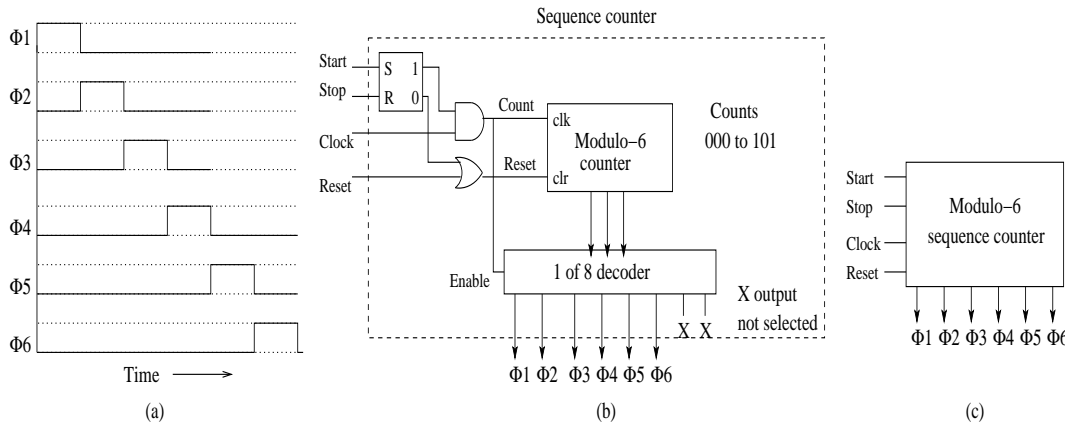


Figure 8: Modulo-6 sequence counter

A modulo-6 counter counts from 000 to 101 and then sets to 000.

Modulo-6 sequence counter consists of mod-6 counter + 1-of-8 decoder (Fig. 8).

Most significant two outputs of 1-of-8 decoder remain deactivated (X) all the time.

If a memory access requires two clock period, then the CU needs a modulo-8 sequence counter (in an iteration, maximum number of memory accesses is 2).

Φ_i s effectively divide the time required for one complete cycle by k (6) equal parts.

Input lines Start/Stop and a flip-flop of Figure 8 are to turn the counter on and off.

A pulse on Start line causes counter to start counting its states.

A pulse on Stop line disconnects clock and resets the counter.

Design: Consider the single addressed CPU with following 7 macro instructions.

<i>Mnemonic</i>	<i>Description</i>
Load X	$AC \leftarrow M(X)$
Store X	$M(X) \leftarrow AC$
Add X	$AC \leftarrow AC + M(X)$
And X	$AC \leftarrow AC \wedge M(X)$
Jump X	$PC \leftarrow X$ (unconditional branch)
Jumpz X	if $AC = 0$ then $PC \leftarrow X$ (conditional branch)
Comp	$AC \leftarrow AC'$ (complement accumulator)

The control signals that execute all the micro instructions are

<i>Control signal</i>	<i>Operation controlled</i>
c_0	$AR \leftarrow PC$
c_1	$DR \leftarrow M(AR)$ <i>Read Memory</i>
c_2	$PC \leftarrow PC + 1(k)$
c_3	$IR \leftarrow DR(Opcode)$
c_4	$AR \leftarrow DR(address)$
c_5	$AC \leftarrow DR$
c_6	$DR \leftarrow AC$
c_7	$M(AR) \leftarrow DR$ <i>Write Memory</i>
c_8	$AC \leftarrow AC + DR$
c_9	$AC \leftarrow AC \wedge DR$
c_{10}	$PC \leftarrow DR(address)$
c_{11}	$AC \leftarrow AC'$ <i>Complement Accumulator</i>

Figure 3 describes instruction fetch cycle.

Execution of a macro-instruction is then 6-step operation as depicted in Figure 7.

Figure 9 is the general structure of hardwired CU.

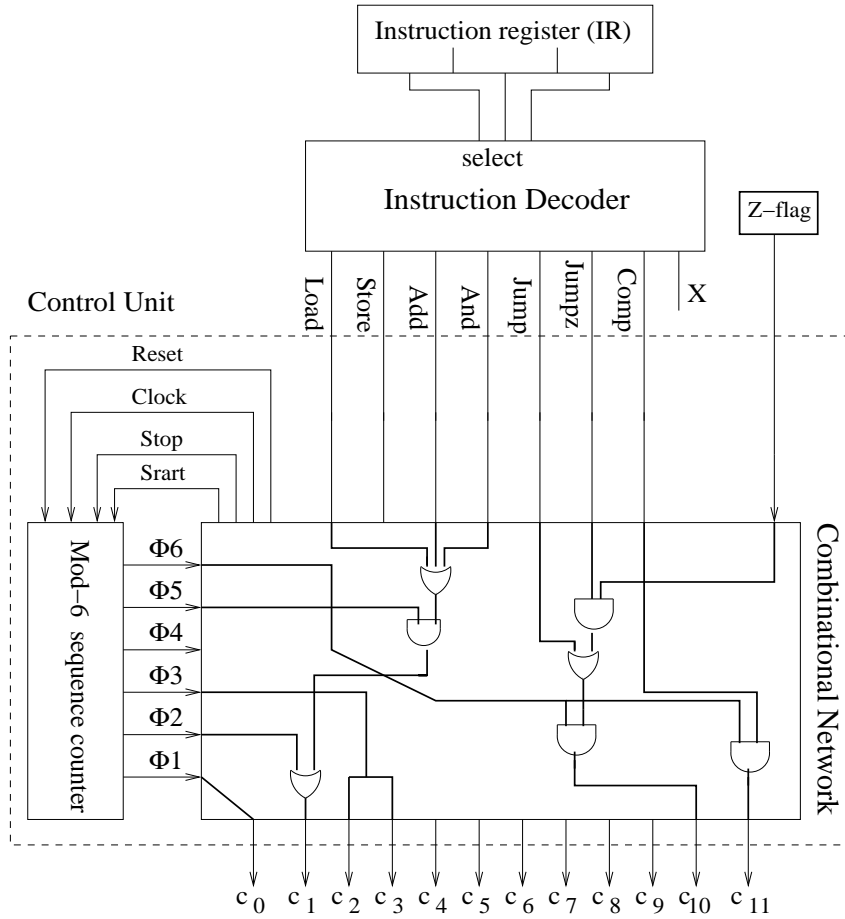


Figure 9: Hardwired CPU control unit around a sequence counter

Example: c_1 which causes a memory read, is activated when $\Phi_2 = 1$.

It is also activated when $\Phi_5 = 1$ during operand fetching for Load, Add, or And.

Therefore, c_1 can be defined as (follow Figure 10)

$$c_1 = \Phi_2 + \Phi_5 (\text{Load} + \text{Add} + \text{And})$$

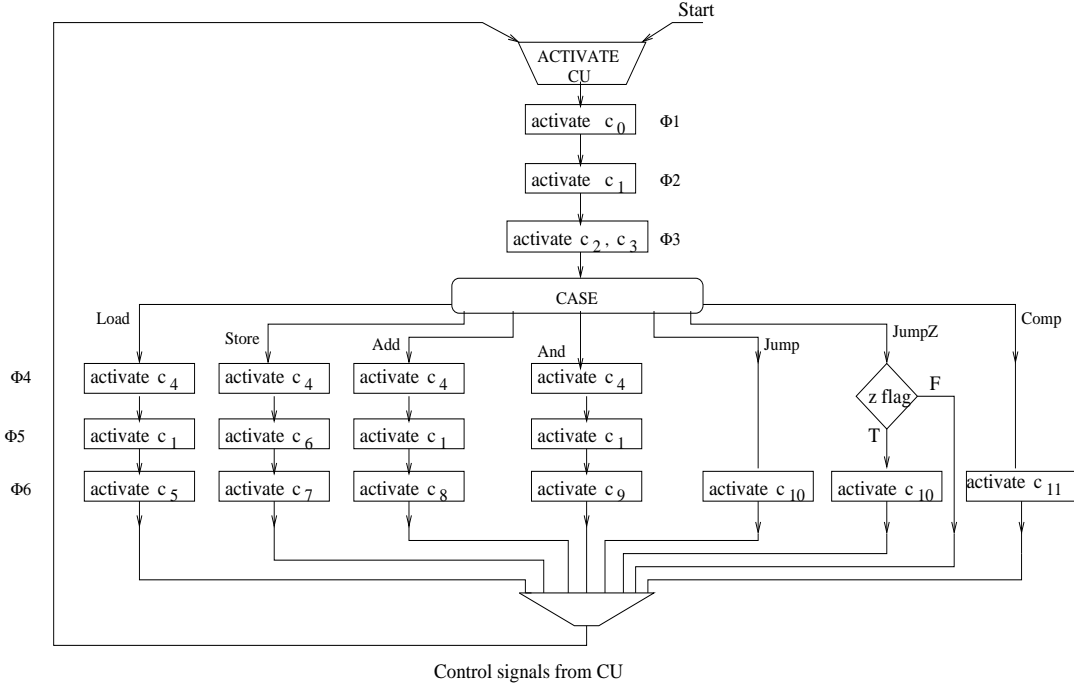


Figure 10: Control signal

The control signal c_i can be defined as:

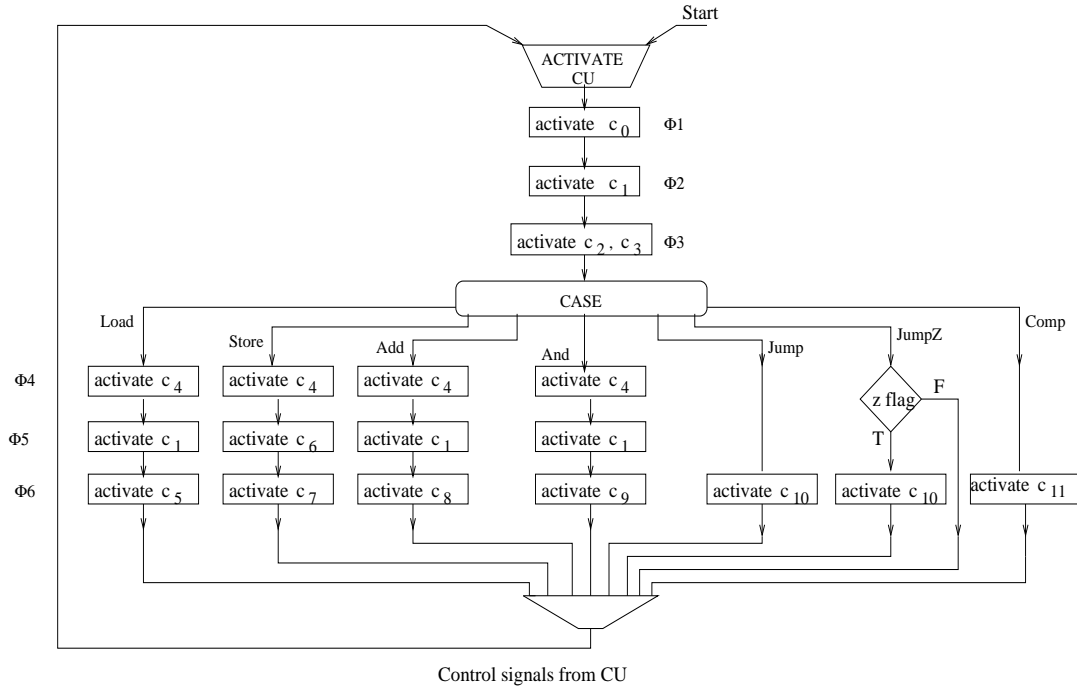
$$\begin{aligned}
 c_0 &= \Phi_1 \\
 c_1 &= \Phi_2 + \Phi_5(\text{Load} + \text{Add} + \text{And}) \\
 c_2 &= \Phi_3 \\
 c_3 &= \Phi_3 \\
 c_4 &= \Phi_4 (\text{Load} + \text{Store} + \text{Add} + \text{And}) \\
 c_5 &= \Phi_6 \cdot \text{Load} \\
 c_6 &= \Phi_5 \cdot \text{Store} \\
 c_7 &= \Phi_6 \cdot \text{Store} \\
 c_8 &= \Phi_6 \cdot \text{Add} \\
 c_9 &= \Phi_6 \cdot \text{And} \\
 c_{10} &= \Phi_6 (\text{Jump} + \text{Jumpz} \cdot \text{z-flag}) \\
 c_{11} &= \Phi_6 \cdot \text{Comp}
 \end{aligned}$$

Thus the combinational circuit of Figure 9 is to be such that the c_i s are realized.

Hardwired CU design is inflexible. It is extremely difficult to introduce any modifications and requires a huge design effort even for a moderate change in design.

Example: Assume introduction of a new instruction *Clear*.

Micro-instructions to be executed for *Clear* macro-instruction are

$$\begin{aligned} DR &\leftarrow AC &< c_6 >, \\ AC &\leftarrow AC' &< c_{11} >, \\ AC &\leftarrow AC \wedge DR &< c_9 >. \end{aligned}$$


CU of new CPU with 8 macro-instructions (7 + *Clear*), has also 12 control signals.

Modification needed is the change in combinational network.

Consider X output of instruction decoder (Figure 9) denotes *Clear*.

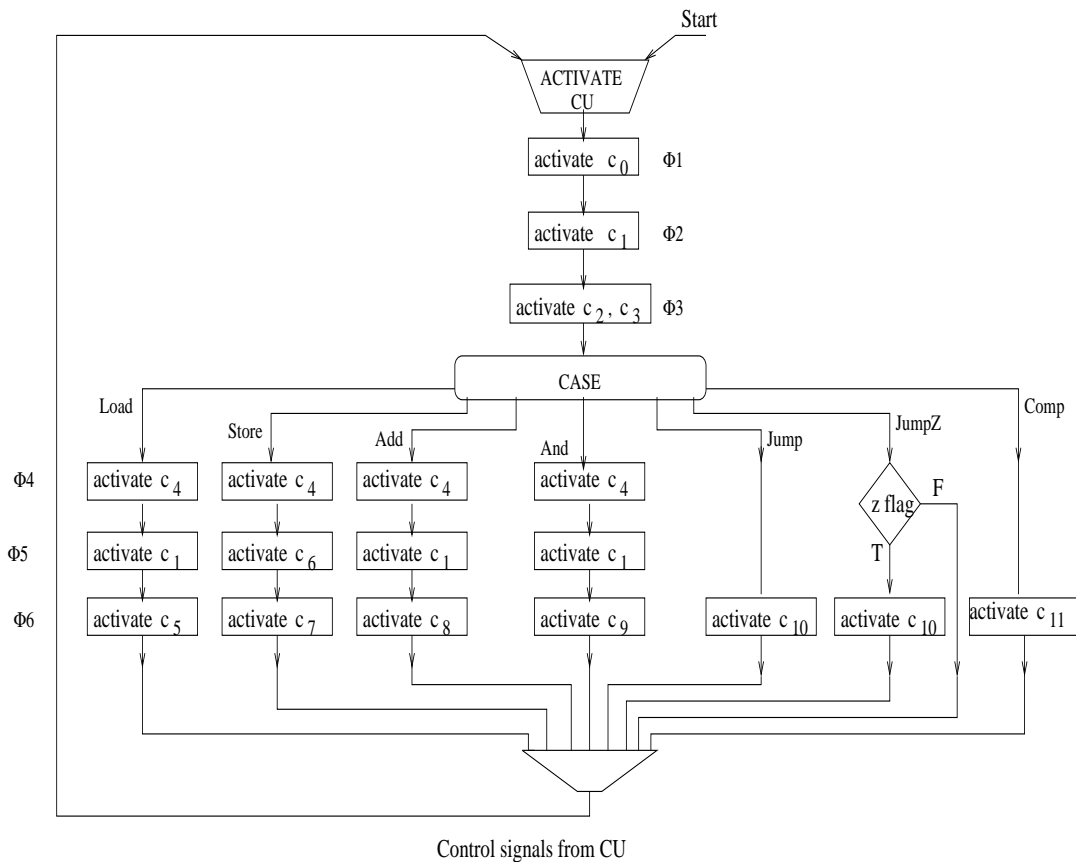
Then expressions for control signals c_6 and c_{11} are to be modified as

$$\begin{aligned} c_6 &= \Phi_5.\text{Store} + \Phi_4.\text{Clear} \text{ [in unmodified } c_6 = \Phi_5 \text{ Store]}, \\ c_{11} &= \Phi_6.\text{Comp} + \Phi_5.\text{Clear} \text{ [in unmodified } c_{11} = \Phi_6 \text{ Comp]}, \\ c_9 &= \Phi_6 (\text{And} + \text{Clear}) \text{ [in unmodified } c_9 = \Phi_6 \text{ And}]. \end{aligned}$$

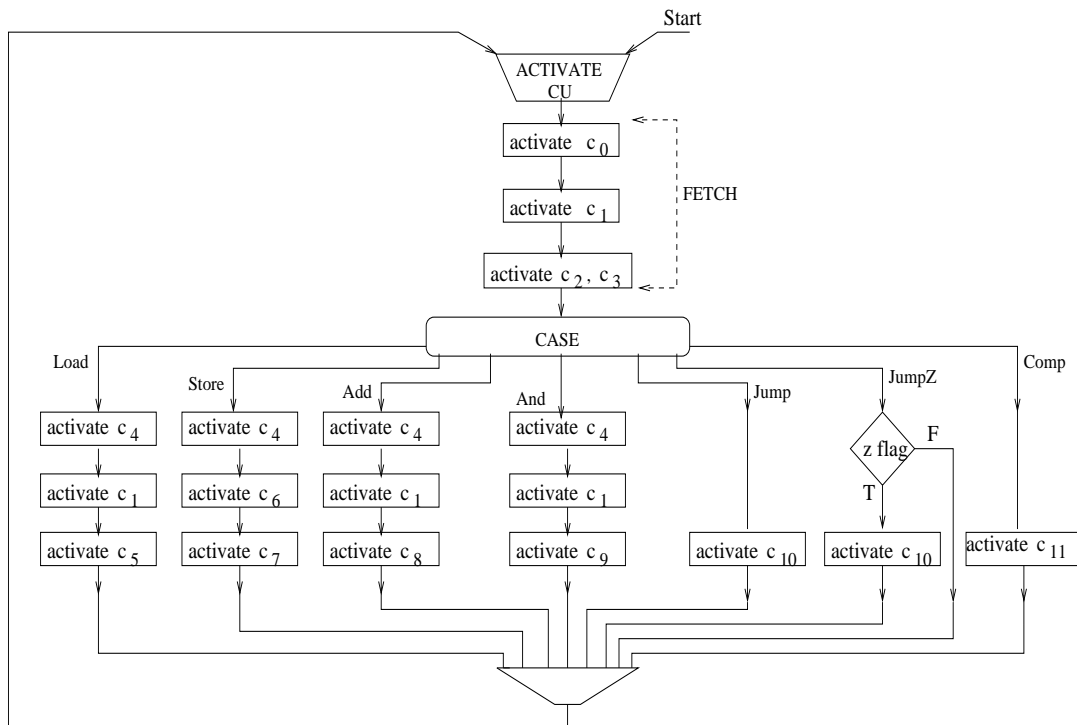
Such a minor change in logic expression may demand massive design effort to resolve different issues (routing/placement/ power dissipation) almost afresh.

0.3 Microprogrammed Control Design

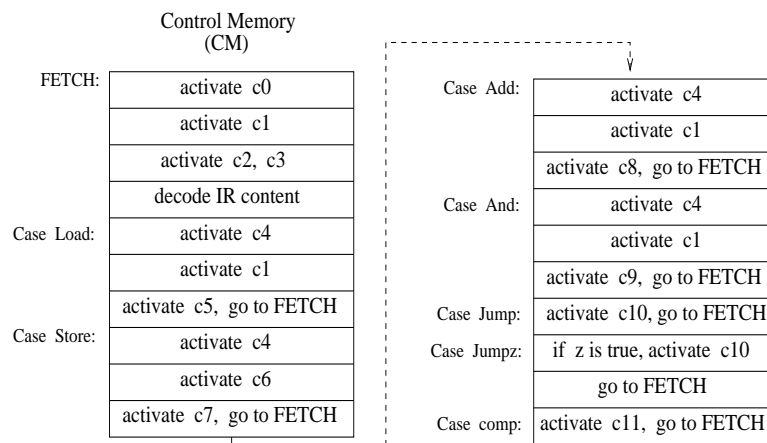
Target is to generate the sequence of control signals as noted in following figure.



In a microprogrammed control design this can be achieved by executing a set of microinstructions (microprogramme) as shown in Figure 11(b).



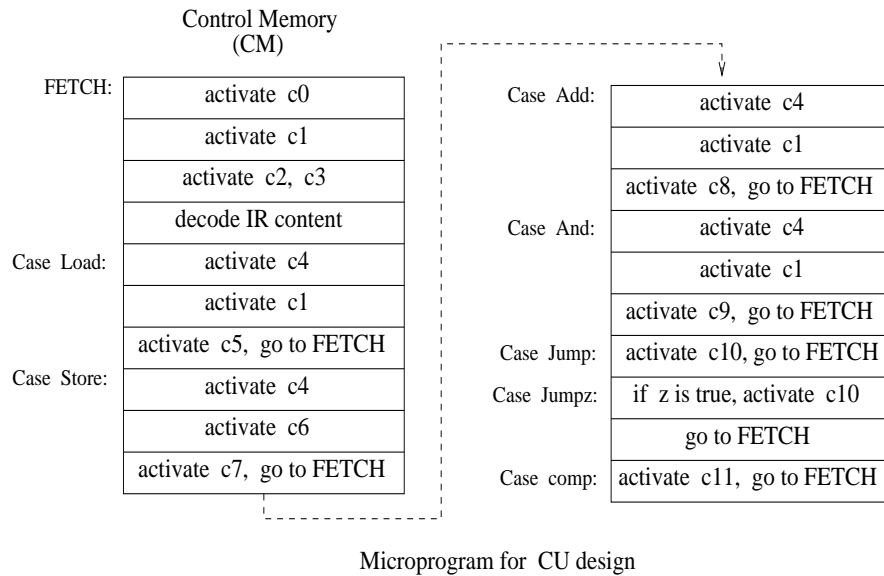
(a) Control signals from micro programmed CU



(b) Microprogram for CU design

Figure 11: Overview of micro-programmed control design

Microinstructions are stored in a special class of memory - control memory (CM).



Content of CM describes that the if the CPU is switched on, CU can fetch μ -instruction *activate c₀* and execute.

It ensures tranfer of PC content to AR.

Then CU fetches and executes *activate c₁* that realizes memory read $DR \leftarrow M(AR)$.

That is, execution of the set of microinstructions stored im CM realizes CU function.

The design technique demands attention to the following issues:

1. Encoding of μ -instructions.
2. μ -instruction sequencing.

After activation of *c₀* it is to be ensured that *activate c₁* is fetched from CM.

3. Address mapping -once macroinstruction fetched from MM is found *Store* (say), then CU must fetch μ -instruction from address *case Store* (Figure 11(b)).

That is, depending on content of IR the CU should decide on next executable μ -instruction address.

0.3.1 Micro-instruction encoding

CU fetches an instruction to a register called control memory data register (CMDR) or microinstruction register (μ IR) (Figure 12).

Decoding of microinstruction is required to activate one (*activate* c_0) or multiple (*activate* c_2, c_3) control signals.

Outputs of the decoder (control signals) are input to different parts of CPU.

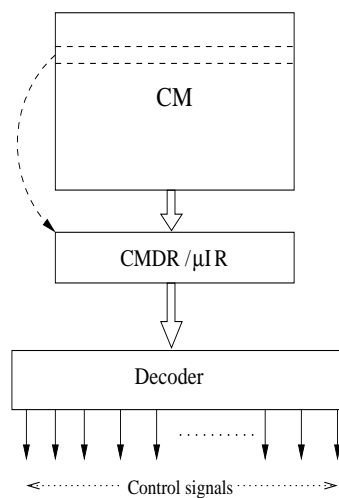


Figure 12: Microinstruction decoding

Encoding controls size of a CM.

Encoding scheme can reduce the CM word size but may increase delay in decoding μ -instructions as well as limit possibility of parallel activation of control signals.

Three encoding schemes are considered for μ -programmed CU instruction format.

1. Horizontal
2. Vertical
3. Diagonal

0.3.1.1 Horizontal format

Horizontal μ -instruction encoding for CU of Figure 11(a)) is in Figure 13(a).

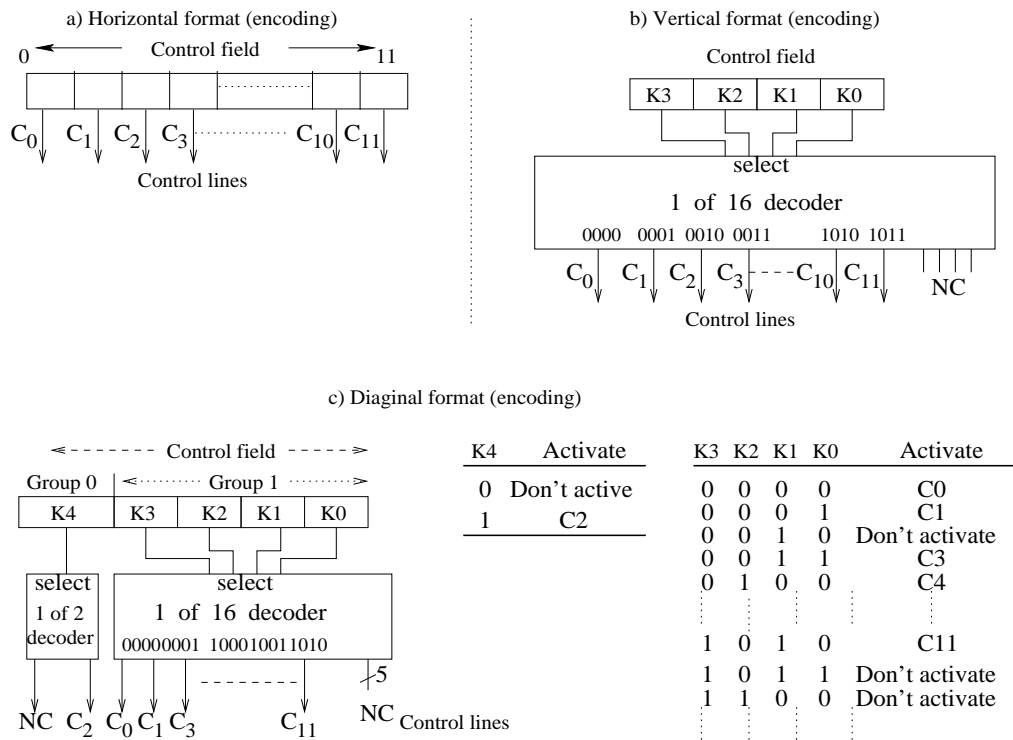


Figure 13: Microinstruction formats

It assigns one bit per control signal.

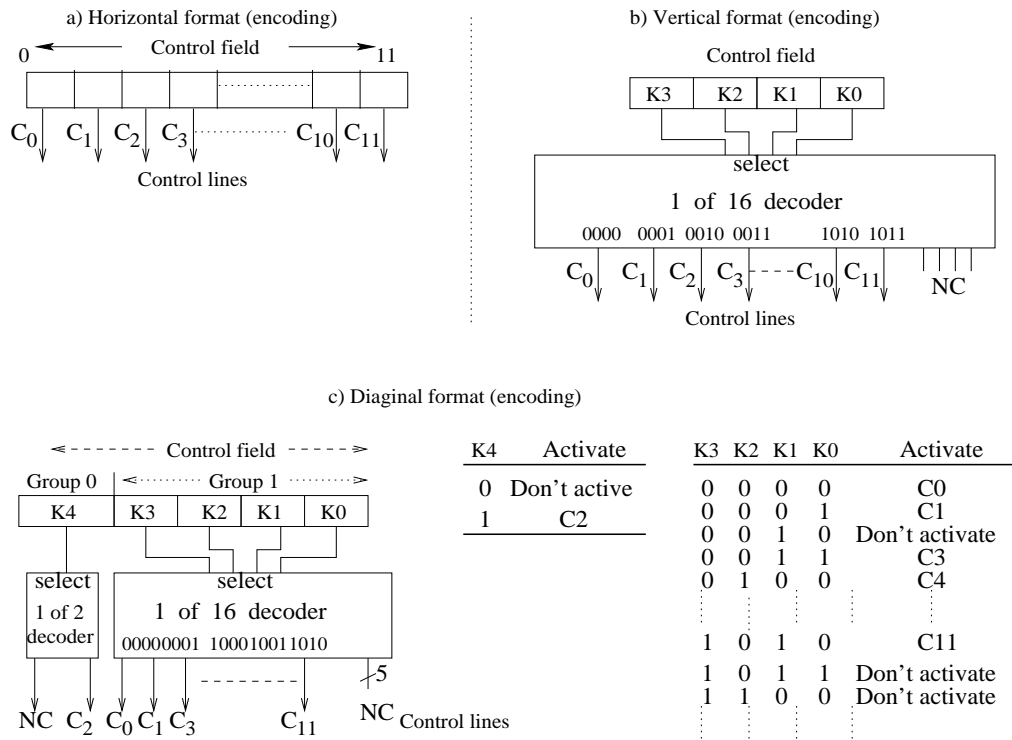
In example design, it requires 12-bit field (control field).

Horizontal format in general does not require decoder - avoids delay of decoding.

The features of horizontal format -

- It is a long format,
- It has ability to express high degree of parallelism,
- It considers little encoding (generally no encoding) of control information.

0.3.1.2 Vertical format



Features of a vertical format (Figure 13(b)) are

- a) This is of short format,
- b) Limited ability to support parallelism (generally no parallelism) in μ -operations,
- c) Accepts considerable encoding of control information.

For a CU, realized with n control signals, this demands m -bit, where $n \leq 2^m$.

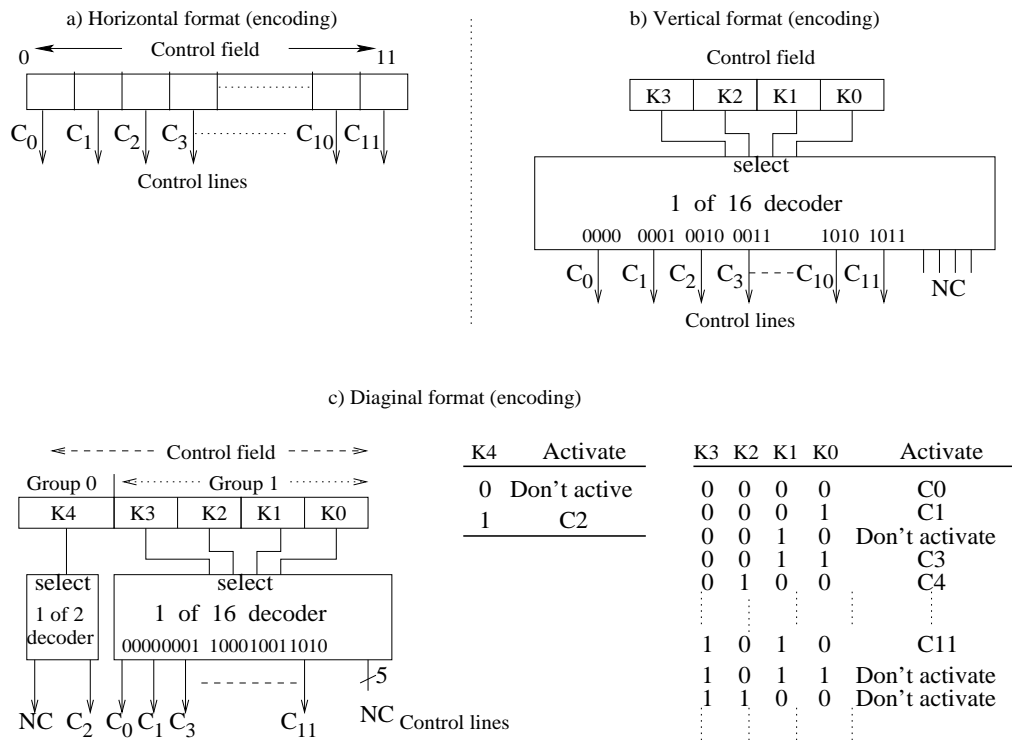
For $n=12$ control signals of example design, we need $m=4$ bits.

The 4-bit (K3 K2 K1 K0) code is then decoded by a 1-of-16 decoder.

Each output of decoder is connected to a control signal (Figure 13(b)).

For example, if K3 K2 K1 K0 = 1010, then c_{10} is activated.

0.3.1.3 Diagonal format



Limitation of vertical format - a microinstruction can't generate two control signals.

That is, parallel execution of microoperations are not allowed.

In example design, two control signals c_2 and c_3 can be considered in parallel.

Vertical encoding can't allow this. Solution is: c_2 and c_3 are generated sequentially.

Alternative solution is: n μ -instructions are partitioned into m groups to get maximum m parallel microoperations with in a μ -instruction.

Each group is encoded as in vertical format.

To decode a μ -instruction, a decoder is required for each group.

In Figure 13(c): encoding of control signals in $m = 2$ groups (Group 0 and Group).

Three decoders are needed to generate the control signals.

Degree of parallelism (parallel execution of μ -operations) offered is $m = 3$.

0.3.1.4 Microinstruction sequencing

There are two options for instruction sequencing

1. Include next microinstruction μI_{i+1} address with in the microinstruction μI_i .
2. Microprogram counter (μPC) to store next executable μ -instruction address.

We assume μ -instruction sequencing with μPC .

0.3.2 Microinstruction address mapping

Shown in Figure 14.

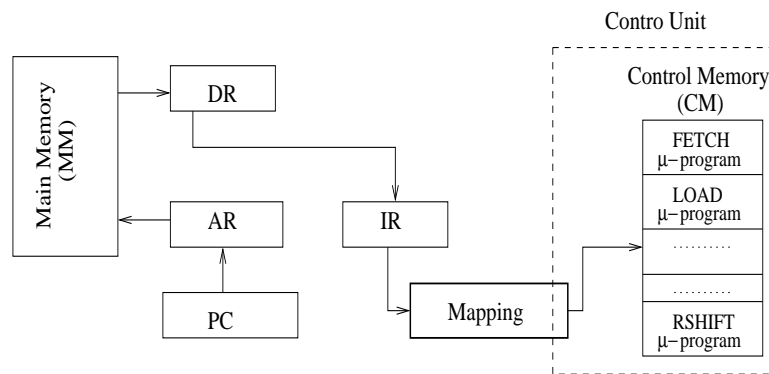


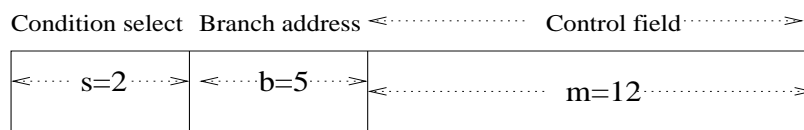
Figure 14: Micro instruction mapping

0.3.3 Microprogrammed control unit

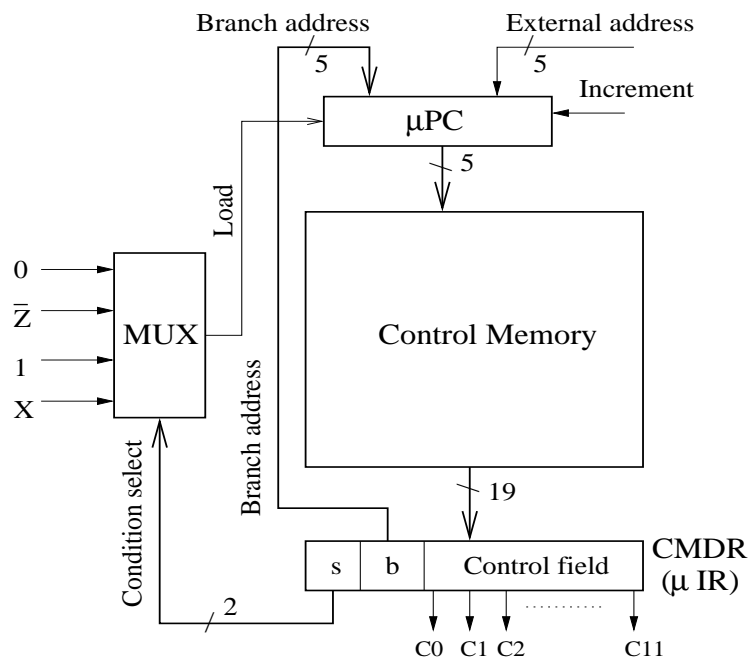
Shown in Figure 15.

Assume horizontal format.

CM size is 32 word. Each of 19 bits. 12-bit is for control field, 5-bit for branch address, and 2-bit for condition select.



a) Micro-instruction format



b) Micro-programmed control unit structure

Figure 15: Micro programmed control unit

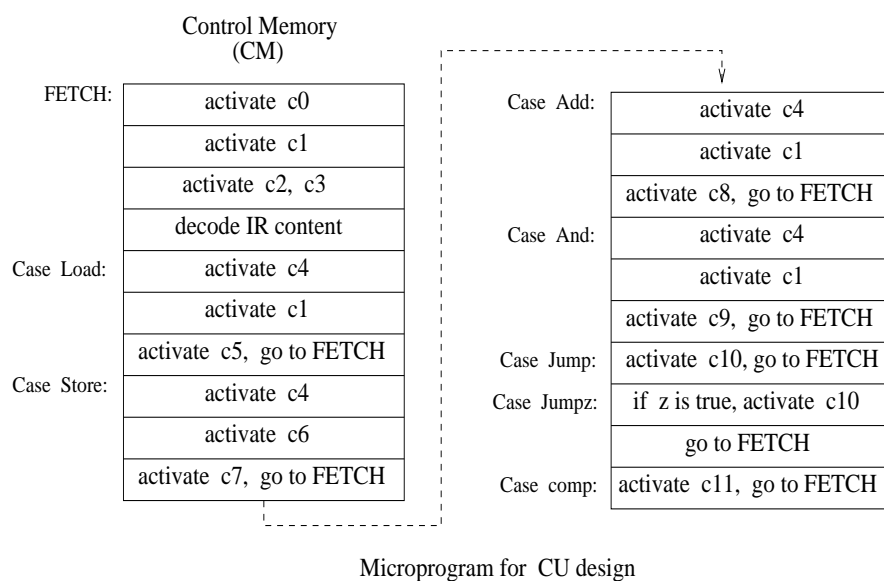
Lecture 16-17: March 19, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

0.4 Example Design - Microprogrammed CU

Let consider the design of CU for the CPU with 7 macro instructions.



Consider FETCH is a macro-instruction. Now, assume opcode for

<i>FETCH</i>	000
<i>LOAD</i>	001
<i>STORE</i>	010
<i>ADD</i>	011
<i>AND</i>	100
<i>JUMP</i>	101
<i>JUMPZ</i>	110
<i>COMP</i>	111

Maximum number of micro-instructions required for a micro-program is 4.

For FETCH it is $3 + 1$ go to IR (decode IR content).

We keep 4 words of CM per micro-program.

That is, CM is having $4 \times 8 = 32$ words.

A word then can be referred by 5-bit addresses.

We apply horizontal formatting.

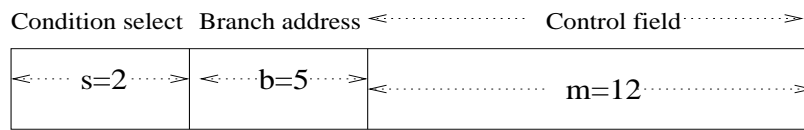
That is, control field is with 12 bits corresponding to 12 control signals.

Considering all the micro-programs - it can be found that there are 4 branch conditions - No branch, Branch always (go to FETCH), conditional branch (if Z is true) and go to IR.

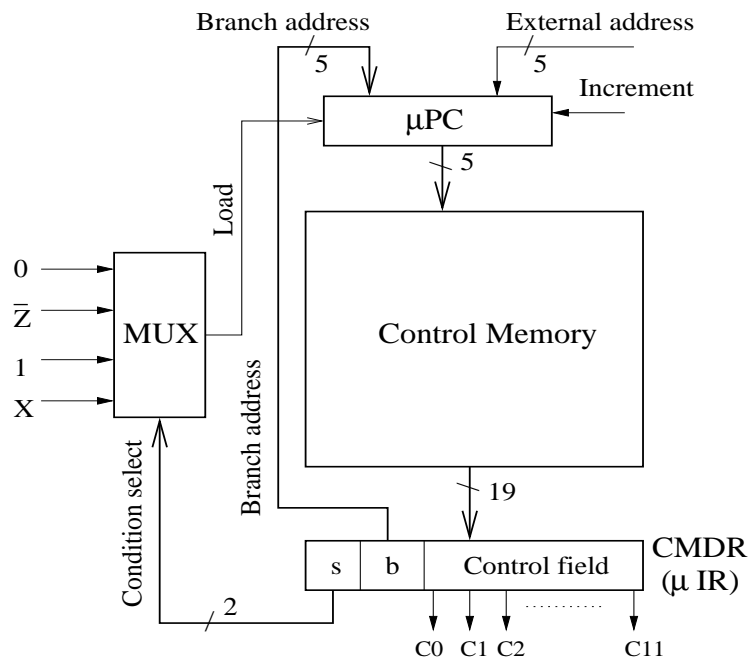
That is, condition select $s = 2$ bits and we need a 4 to 1 MUX¹,

That is, each word of CM is of $2 + 5 + 12 = 19$ bits.

¹When $s = 11$ in Figure 15, X input is passed to the output of MUX. It is assumed that this realizes go to IR (-that is, the content of IR with two augmented 0s is loaded to μ PC).



a) Micro-instruction format



b) Micro-programmed control unit structure

Go to IR:

If content of IR is 110, then go to 11000 (opcode + 2 augmented 0s with LSB).

It defines address mapping: the starting address of a micro-program corresponding to a macro-instruction is "opcode + 2 augmented 0s with LSB".

Content of control memory

<i>Macro</i>	<i>Address</i>	<i>Condition select</i>	<i>Branch address</i>	$c_0c_1c_2$	$c_3c_4c_5$	$c_6c_7c_8$	$c_9c_{10}c_{11}$
<i>FETCH</i>	00000	00	00000	100	000	000	000
	00001	00	00000	010	000	000	000
	00010	00	00000	001	100	000	000
	00011	11	00000	<i>go</i>	<i>to</i>	<i>IR</i>	
<i>LOAD</i>	00100	00	00000	000	010	000	000
	00101	00	00000	010	000	000	000
	00110	10	00000	000	001	000	000
	00111						
<i>STORE</i>	01000	00	00000	000	010	000	000
	01001	00	00000	000	000	100	000
	01010	10	00000	000	000	010	000
	01011						
<i>ADD</i>	01100	00	00000	000	010	000	000
	01101	00	00000	010	000	000	000
	01110	10	00000	000	000	001	000
	01111						
<i>AND</i>	10000	00	00000	000	010	000	000
	10001	00	00000	010	000	000	000
	10010	10	00000	000	000	000	100
	10011						
<i>JUMP</i>	10100	10	00000	000	000	000	010
	10101						
	10110						
	10111						
<i>JUMPZ</i>	11000	01	00000	000	000	000	000
	11001	10	00000	000	000	000	010
	11010						
	11011						
<i>COMP</i>	11100	10	00000	000	000	000	001

Example: Execution of the following program segment in MM.

```

      S :  LOAD    L
    S + 1 :  ADD    Y
    S + 2 :  JUMPZ  X
      ⋮
    X :  ...      ...
      ⋮
    L :  05
      ⋮
    Y :  13

```

1. Set $PC \leftarrow S$, then *execute* $\Rightarrow \mu PC \leftarrow 00000$.
2. Location 00000 of CM is fetched - c_0 is activated $\Rightarrow AR \leftarrow S, \mu PC \leftarrow 00001$, no branch.
3. Location 00001 of CM is fetched - c_1 is activated $\Rightarrow DR \leftarrow 001 L, \mu PC \leftarrow 00010$, no branch.
4. Location 00010 of CM is fetched - c_2 and c_3 are activated $\Rightarrow PC \leftarrow S+1, IR \leftarrow 001, \mu PC \leftarrow 00011$, no branch.
5. Location 00011 of CM is fetched - $\mu PC \leftarrow 00100$, go to IR $\Rightarrow \mu PC \leftarrow 00100$.
6. Location 00100 of CM is fetched - c_4 is activated $\Rightarrow AR \leftarrow L, \mu PC \leftarrow 00101$, no branch.
7. Location 00101 of CM is fetched - c_1 is activated $\Rightarrow DR \leftarrow 05, \mu PC \leftarrow 00110$, no branch.
8. Location 00110 of CM is fetched - c_5 is activated $\Rightarrow AC \leftarrow 05, \mu PC \leftarrow 00111$, branch always - $\mu PC \leftarrow 00000$.
9. Location 00000 of CM is fetched - c_0 is activated $\Rightarrow AR \leftarrow S+1, \mu PC \leftarrow 00001$, no branch.

.....

Micro-programmed CU is slower than hardwired CU.

It is easy to modify/upgrade.

Example: Assume introduction of a new instruction *CLEAR* in place of ADD.

Micro-instructions to be executed for *CLEAR* macro-instruction are

$$\begin{aligned} \text{DR} &\leftarrow \text{AC} &< c_6 >, \\ \text{AC} &\leftarrow \text{AC}' &< c_{11} >, \\ \text{AC} &\leftarrow \text{AC} \wedge \text{DR} &< c_9 >. \end{aligned}$$

So, opcode for *CLEAR* is 011.

Therefore, content of CM locations 01100 to 01110 are to be modified.

<i>Macro</i>	<i>Address</i>	<i>Condition</i> <i>select</i>	<i>Branch</i> <i>address</i>	$c_0c_1c_2$	$c_3c_4c_5$	$c_6c_7c_8$	$c_9c_{10}c_{11}$
\vdots							
<i>CLEAR</i>	01100	00	00000	000	000	100	000
	01101	00	00000	000	000	000	001
	01110	10	00000	000	000	000	100
	01111						
\vdots							

0.5 Control Optimization

Length of each micro-program depends on

- a. Complexity of micro-instruction, and
- b. Parallelism of micro-operations on the available data path.

Target of control optimization is -

Reduce number of words H of CM, and number of bits W per word.

0.5.1 Reducing H

H depends on individual micro-program length.

H is reduced by including parallel micro-operations with in a single micro-instruction.

0.5.2 Minimizing W

CU encodes control signals corresponding to the micro-operations in a field/group in such a way that no two signals encoded in this field are activated simultaneously in any micro-instruction -that is, these should be mutually exclusive.

In example design, c_2 and c_3 are to be placed in two different groups.

Two control signals c_i and c_j ($i \neq j$) are called mutually exclusive (also called compatible) if $c_i \in I_k$, then $c_j \notin I_k$ for all k .

A set of control signals $C_r = \{c_i, c_j, c_k, \dots\}$ with pairwise compatibility among $\{c_i, c_j, c_k, \dots\}$ is a compatibility class.

All these control signals can be encoded in the same control field/group.

If number of such control signals in C_r is N_r , then number of bits in control field of a micro-instruction is

$$W = \sum_k \lceil \log_2(N_r + 1) \rceil.^2$$

k denotes number of compatibility classes.

That is, target of reducing W effectively leads to target of reducing k .

This can be realized by increasing cardinality ($|N_r|$) of each compatibility class.

²+1 is to encode the don't activate a control signal option.

The following steps can be followed to reduce W.

Step 1 : Find maximal compatibility classes (MCCs) of control signals.

Step 2 : Determine all minimal set of MCCs that include each of control signals.
Each of these sets is the minimal MCC cover.

Step 3 : a) Check each MCC cover to determine all ways of including each control signal exactly in one MCC (C_r).

b) Compute W of each such resulting solution.

c) Select one with minimum W.

Example 0.1 Minimize W to encode control signals of micro-instructions -

<i>Microinstruction</i>	<i>Control signals</i>
I_1	c_1, c_2, c_3, c_7
I_2	c_1, c_3, c_5, c_8
I_3	c_1, c_4, c_6
I_4	c_2, c_3, c_6

Step 1: Computation of the maximal compatibility classes (MCCs).

$S_1: \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8\}$

$S_2: \{c_2c_4, c_2c_5, c_2c_8, c_3c_4, c_4c_5, c_4c_7, c_4c_8, c_5c_6, c_5c_7, c_6c_7, c_6c_8, c_7c_8\}$

$S_3: \{c_2c_4c_5, c_2c_4c_8, c_4c_5c_7, c_4c_7c_8, c_5c_6c_7, c_6c_7c_8\}$

$S_4: \{\phi\}$.

The 8 MCCs are $MC_1 = \{c_1\}$, $MC_2 = \{c_3, c_4\}$, $MC_3 = \{c_2, c_4, c_5\}$, $MC_4 = \{c_2, c_4, c_8\}$, $MC_5 = \{c_4, c_5, c_7\}$, $MC_6 = \{c_4, c_7, c_8\}$, $MC_7 = \{c_5, c_6, c_7\}$ and $MC_8 = \{c_6, c_7, c_8\}$.

Step 2: Extract minimal MCC covers.

To identify minimal MCC covers, construct cover table (Table 1).

A row is for MC_i and column is for control signal c_j .

X entry denotes - $c_j \in MC_i$ -that is, MC_i covers c_j .

A bold **X** in an MC denotes MC is the essential MCC.

In Table 1, essential MCCs are the MC_1 and MC_2 .

Table 1: Cover table

MCCs	Control signals							
	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
$MC_1 = c_1$	X							
$MC_2 = c_3c_4$			X	X				
$MC_3 = c_2c_4c_5$		X		X	X			
$MC_4 = c_2c_4c_8$		X		X				X
$MC_5 = c_4c_5c_7$				X	X		X	
$MC_6 = c_4c_7c_8$				X			X	X
$MC_7 = c_5c_6c_7$					X	X	X	
$MC_8 = c_6c_7c_8$						X	X	X

The next tasks are -

- a. Delete essential MCCs (MC_1 and MC_2) from cover table - Column c_1 and c_3 are deleted.

As MC_2 includes c_4 , column c_4 can also be deleted (Table 2).

MCCs	Control signals							
	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
$MC_1 = c_1$	X							
$MC_2 = c_3c_4$			X	X				
$MC_3 = c_2c_4c_5$		X		X	X			
$MC_4 = c_2c_4c_8$		X		X				X
$MC_5 = c_4c_5c_7$				X	X		X	
$MC_6 = c_4c_7c_8$				X			X	X
$MC_7 = c_5c_6c_7$					X	X	X	
$MC_8 = c_6c_7c_8$						X	X	X

Table 2: Cover table after elimination of essential rows and corresponding columns

MCCs	Control signals				
	c_2	c_5	c_6	c_7	c_8
$MC_3 = c_2c_4c_5$	X	X			
$MC_4 = c_2c_4c_8$	X				X
$MC_5 = c_4c_5c_7$		X		X	
$MC_6 = c_4c_7c_8$				X	X
$MC_7 = c_5c_6c_7$		X	X	X	
$MC_8 = c_6c_7c_8$			X	X	X

- b. If cover table contains two or more identical columns, then all but one of those columns can be deleted.

In the current example, no such case.

c. Delete dominating column c_i -

c_i dominates c_j if c_i contains an X in every row where c_j contains an X and c_i contains more Xs than c_j .

Dominating column c_i is deleted as the MCC that covers c_j also covers c_i .

In the current example, c_7 dominates c_6 (Table 3).

MCCs	Control signals				
	c_2	c_5	c_6	c_7	c_8
$MC_3 = c_2c_4c_5$	X	X			
$MC_4 = c_2c_4c_8$	X				X
$MC_5 = c_4c_5c_7$		X		X	
$MC_6 = c_4c_7c_8$				X	X
$MC_7 = c_5c_6c_7$		X	X	X	
$MC_8 = c_6c_7c_8$			X	X	X

Table 3 is after eliminating dominating column.

Table 3: Cover table after elimination of dominating columns

MCCs	Control signals			
	c_2	c_5	c_6	c_8
$MC_3 = c_2c_4c_5$	X	X		
$MC_4 = c_2c_4c_8$	X			X
$MC_5 = c_4c_5c_7$		X		
$MC_6 = c_4c_7c_8$				X
$MC_7 = c_5c_6c_7$		X	X	
$MC_8 = c_6c_7c_8$			X	X

d. Delete dominated row MC_j -

MC_i dominates MC_j if MC_i contains X in every col where MC_j is having.

Dominated row MC_j is deleted since MC_i covers all control signals that are covered by MC_j .

In example, MC_3 dominates MC_5 , MC_4 dominates MC_6 , MC_7 dominates MC_5 , and MC_8 dominates MC_6 . Dominated rows are MC_5 and MC_6 .

MCCs	Control signals			
	c_2	c_5	c_6	c_8
$MC_3 = c_2c_4c_5$	X	X		
$MC_4 = c_2c_4c_8$	X			X
$MC_5 = c_4c_5c_7$		X		
$MC_6 = c_4c_7c_8$				X
$MC_7 = c_5c_6c_7$		X	X	
$MC_8 = c_6c_7c_8$			X	X

Cover table after removal of dominated rows is shown in Table 4.

Table 4: Cover table after elimination of dominated rows

MCCs	Control signals			
	c_2	c_5	c_6	c_8
$MC_3 = c_2c_4c_5$	X	X		
$MC_4 = c_2c_4c_8$	X			X
$MC_7 = c_5c_6c_7$		X	X	
$MC_8 = c_6c_7c_8$			X	X

Minimal cover for Table 4 contains two MCCs and possible choices are

$\{MC_3, MC_8\}$ and $\{MC_4, MC_7\}$.

Therefore, desired MCC solutions are

$\{MC_1, MC_2, MC_3, MC_8\}$ and $\{MC_1, MC_2, MC_4, MC_7\}$.

Step 3: Find minimum cost W.

- a. In minimal MCC cover $\{MC_1, MC_2, MC_3, MC_8\}$, each of the control signals $c_1, c_2, c_3, c_5, c_6, c_7$ and c_8 can be covered by only one MCC (Table 5).

Table 5: Cover table to find cost

MCCs	Control signals							
	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
$MC_1 = c_1$	X							
$MC_2 = c_3c_4$			X	X				
$MC_3 = c_2c_4c_5$		X		X	X			
$MC_8 = c_6c_7c_8$						X	X	X

Control signal c_4 can be covered by either MC_2 or MC_3 (Table 5).

- i. Let c_4 is included in MC_2 - MCC cover include partitions as $\{\{MC_1 = c_1\}, \{MC_2 = c_3c_4\}, \{MC_3 = c_2c_5\}, \{MC_8 = c_6c_7c_8\}\}$.

That is, 4 groups in diagonal format.

It requires 1-bit for first group MC_1 (to activate c_1 or don't activate),

2-bit for second group MC_2 (c_3c_4), 2-bit for third MC_3 (c_2c_5) and

2-bit for fourth group MC_8 ($c_6c_7c_8$).

That is, cost of control field is

$$W = 1+2+2+2 = 7.$$

- ii. Let c_4 is included in MC_3 - MCC cover include partitions as $\{\{MC_1 = c_1\}, \{MC_2 = c_3\}, \{MC_3 = c_2c_4c_5\}, \{MC_8 = c_6c_7c_8\}\}$.

For this, cost is

$$W = 1+1+2+2 = 6.$$

- b. Similarly, for other MCC cover $\{MC_1, MC_2, MC_4, MC_7\}$, there can be two alternative partitions.

For which cost $W=7$ and 6 respectively.

Lecture 18: March 23, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

Memory Subsystem

Memory subsystem is one of the major components of computer system (Figure 1). It includes main memory (MM), secondary memory (SM) and a high speed component of MM (cache).

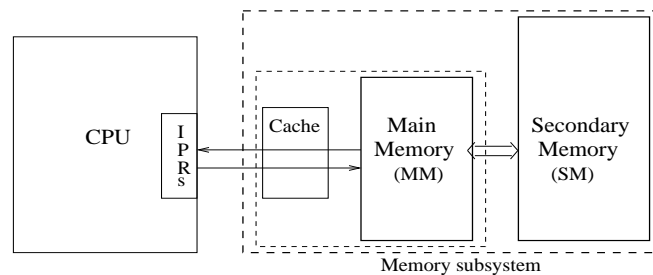


Figure 1: Computer memory subsystem

IPR (internal processor registers) are on-chip highly expensive CPU registers.

This class of memory unit can hold temporary results during computation in progress

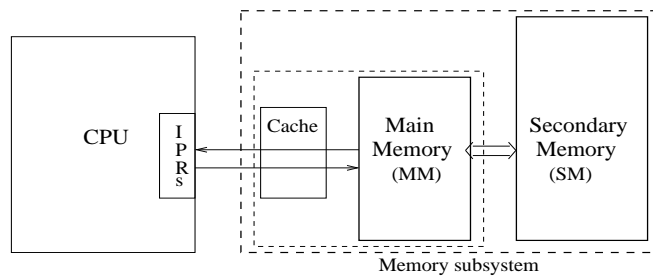
Operates at the CPU speed.

IPRs are very few in numbers due to its excessive cost.

Cache - A special class of high speed main memory conventionally.

It is a product of Bipolar semiconductor technology.

Normally, cache unit is used to store part of a program currently in execution.



Main/primary memory - Programs (part/whole) that are currently in execution reside in MM.

CPU has direct access to MM.

MM is relatively faster and of moderately large capacity.

The cache is also a part of MM.

Secondary memory - Examples of common SMs are

- magnetic disk, optical disk, magnetic tape etc.

SM acts as the backing storage.

CPU cannot directly communicate with SM.

Conventionally, SM is inexpensive, slow and of large capacity than that of MM.

Objective: design of effective memory subsystem to achieve benefits in terms of

- Access time: It is a measure of efficient access to memory.

The access time $t_A = t_2 - t_1$, where at t_1 CPU initiates memory read/write request and read/write operation is completed at t_2 .

We define *read access time* (t_{AR}) and *write access time* (t_{AW}).

The declared access time of a memory can be

$$t_A = \frac{t_{AR} + t_{AW}}{2}.$$

t_{AR} - It is measured as $t_4 - t_3$ (Figure 2(a)).

t_{AW} - It is measured as $t_6 - t_5$ (Figure 2(b)).

Normally, $t_{AW} > t_{AR}$. However, in some memories,

$$t_{AR} \simeq t_{AW} = t_A.$$

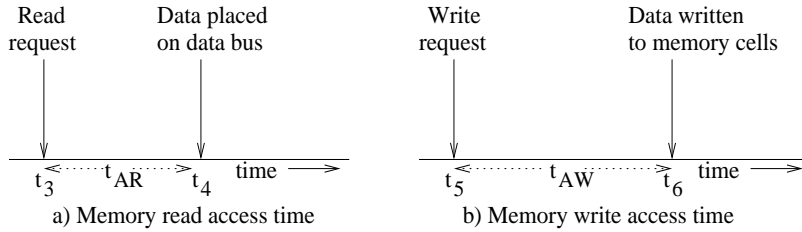


Figure 2: Memory access time

- Access mode: Access time depends also on mode of data access.
 - a. SAM (serial access memory) - In SAM, information can be accessed only in predetermined order.

Access time is - function of location being accessed.

Normally, access time for SAM is proportional to the distance of location of data from a reference position.

Common example of SAM is *magnetic tape*.

SAMs are extremely low cost.

- b. RAM (random access memory) - The read/write access time for RAM is constant - does't depend on location of data.

RAMs are classified as RWM and ROM (Figure 3).

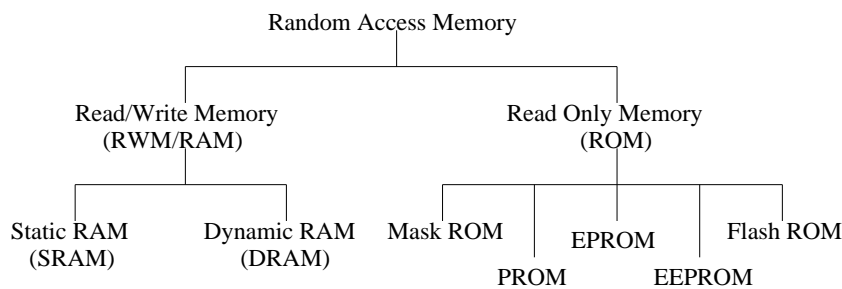


Figure 3: RAM arrays

- c. DAM (direct access memory) - Access time in DAM is semi-random.

It depends on location/position of data item to be accessed but not exactly proportional to the distance. Magnetic hard disk is a DAM.

Access to find an item from disk demands placement of the read/write head on proper track of disc and a rotation of disk to place part of track (sector), containing data item, just under disk head.

Access time = time taken to move disk head + rotation time of disk.

d. CAM (content addressable memory) - It realizes a special form of random access to data items.

An item/word of memory is retrieved based on the part of word's content.

- **Alterability:** It defines whether content of memory can be modified by CPU during program execution.

The read/write memory (RWM/RAM) content is alterable.

Content of conventional ROM cannot be altered during program execution.

- **Permanance of storage:** In a non volatile memory (ROM), information once recorded remains as it is (even if the power is withdrawn) until deliberately changed.

On the other hand, in a volatile memory (RAM), information decays once power is withdrawn.

Information stored in a static RAM (SRAM) can be kept as it is through constant supply of power.

In dynamic RAM (DRAM), information decays even if power is on.

In destructive read out (DRO) memory, if a data is read, data in memory is lost (Figure 4).

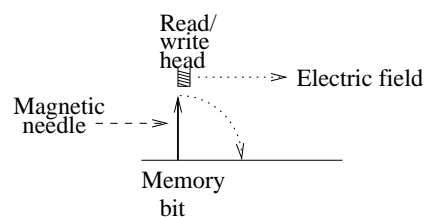


Figure 4: DRO read

The *loss* or *decay* of information adds hardware overhead/delay to CPU instruction execution time (cycle time).

- Cycle time - delay between initiations of two successive memory operations.

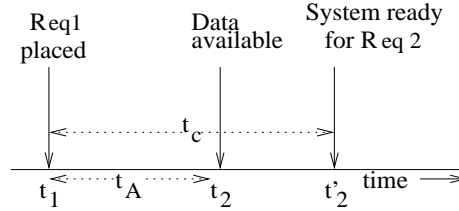


Figure 5: Cycle time

It is - t_A + additional time required before a second access can commence.

Additional time - non-destructive write in DRO, refresh time in DRAM etc.

The cycle time is, therefore,

$$t_c = t'_2 - t_1 \text{ (Figure 5).}$$

In DRAM, if refreshing is required in T_R interval and refresh time is T_r , then

$$t'_2 = t_2 + \frac{T_r}{T_R} \times t_A.$$

In DRO memory, every read operation is followed by a non-destructive write (Figure 6). That is, for DRO memory read,

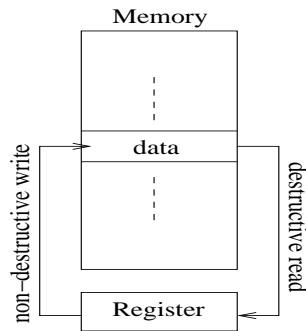


Figure 6: Destructive readout memory

$$t'_2 = t_2 + \text{restoring time required for non-destructive write.}$$

- Cost/bit: Target of memory subsystem design is to reduce cost per bit of memory interfaced.

The cost is due to cost of memory cells, peripheral support/access circuitry and memory refresh/restoring logic.

Cost per bit of B-bit memory unit is - total cost of memory divided by B.

An approximate relation between t_A and cost/bit of memories is in Figure 7.

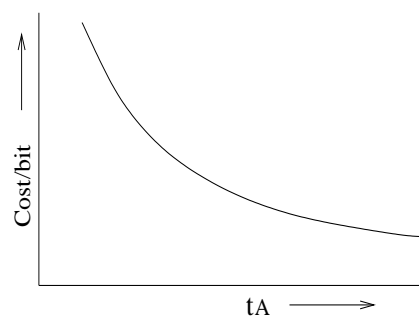


Figure 7: Memory cost and access time

Other than cost of memory, CPU-memory interfacing should be cost effective.

0.1 Memory Interfacing

Basic organization of a memory device is shown in Figure 8.

In addition to power lines, a memory chip consists of -

- (i) m address lines to select one out of 2^m memory locations.
- (ii) d bidirectional data lines for data transfer with CPU.
- (iii) Read/write signal line(s): read (from memory) or write (to memory cells).

In Figure 8, memory module is having only one signal line RD/\overline{WR}

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

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

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

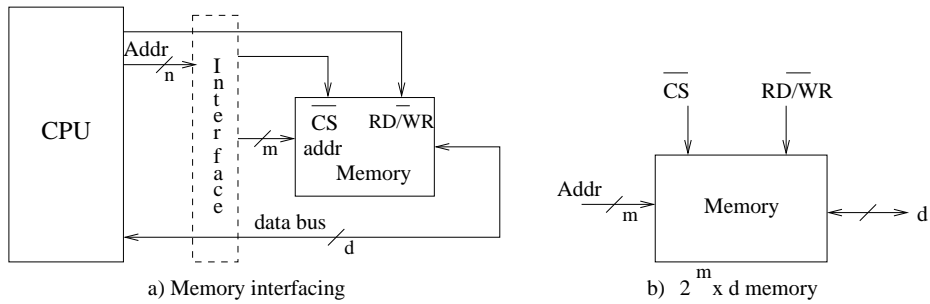


Figure 8: Memory interfacing

Lecture 19-20: March 26, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

Example of memory interfacing

Assume CPU has - 16 address lines ($A_{15}, A_{14}, A_{13}, \dots, A_1, A_0$), 8-bit bidirectional data bus (D), RD/\overline{WR} and IO/\overline{M} to define CPU access: I/O device or memory.

$IO/\overline{M} = 1 \Rightarrow$ access to input/output device.

$IO/\overline{M} = 0 \Rightarrow$ access to memory.

This CPU provides $2^{16} = 64K$ memory address space (0000_h to $FFFF_h$).

It implements I/O mapped I/O - separate address space for memory ($IO/\overline{M} = 0$).

In some earlier processors, the scheme memory mapped I/O was implemented.

Interface a memory (RAM/ROM) chip of 16K×8.

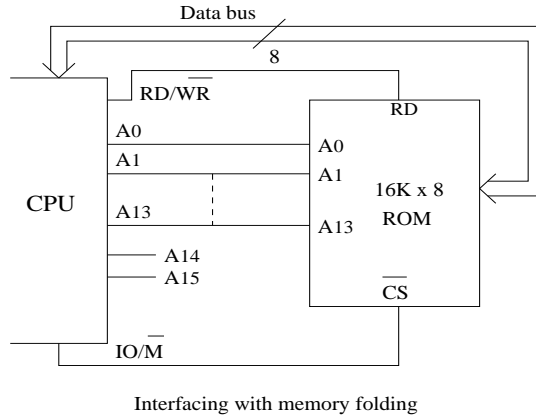


Figure 9: Memory interfacing option 1

Simplest form of interfacing logic is shown in Figure 9.

16K ROM has 14 address lines (A_{13} to A_0). IO/\overline{M} is connected to CS of ROM.

Address space occupied by ROM chip can be

	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	\cdots	A_1	A_0	
1. Address of LS location	0	0	0	0	0	\cdots	0	0	0000_h
Address of MS location	0	0	1	1	1	\cdots	1	1	$3FFF_h$
	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	\cdots	A_1	A_0	
2. Address of LS location	0	1	0	0	0	\cdots	0	0	4000_h
Address of MS location	0	1	1	1	1	\cdots	1	1	$7FFF_h$
	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	\cdots	A_1	A_0	
3. Address of LS location	1	0	0	0	0	\cdots	0	0	8000_h
Address of MS location	1	0	1	1	1	\cdots	1	1	$BFFF_h$
	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	\cdots	A_1	A_0	
4. Address of LS location	1	1	0	0	0	\cdots	0	0	$C000_h$
Address of MS location	1	1	1	1	1	\cdots	1	1	$FFFF_h$

A physical location of memory module can have 4 addresses.

Ex: 4 addresses $0000 \cdots 00$, $0100 \cdots 00$, $1000 \cdots 00$, and $1100 \cdots 00$ point to same physical location. This is memory folding.

Avoiding memory folding

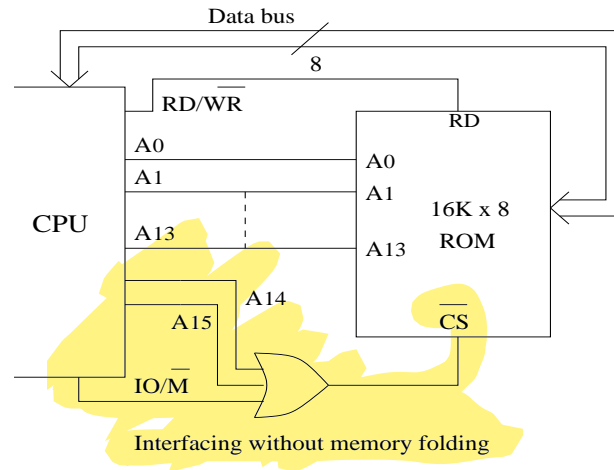


Figure 10: Memory interfacing option 2

Interfacing of Figure 10 avoids memory folding.

Address space occupied by ROM chip is

	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	\dots	A_1	A_0	
<i>Address of LS location</i>	0	0	0	0	0	\dots	0	0	0000h
<i>Address of MS location</i>	0	0	1	1	1	\dots	1	1	3FFFh

This design eliminates provision of any further expansion.

Complete decoding

Effective interfacing technique is complete decoding (Figure 11).

Interfacing logic partitions whole memory address space in small slots.

1 of 4 decoder partitions 64K address space in 4 slots each of 16K.

In Figure 11, only one slot is utilized, the other three are for expansion.

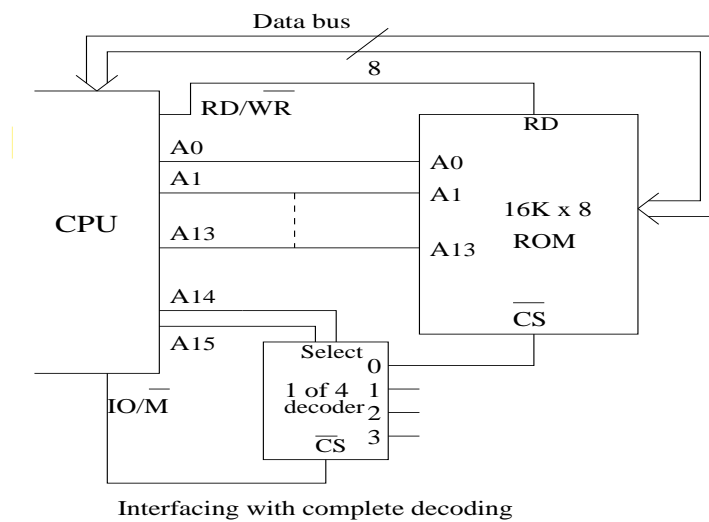


Figure 11: Memory interfacing: complete decoding

Address space occupied by ROM chip is same as Figure 10.

	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	\dots	A_1	A_0	
<i>Address of LS location</i>	0	0	0	0	0	\dots	0	0	0000h
<i>Address of MS location</i>	0	0	1	1	1	\dots	1	1	3FFFh

Expansion in complete decoding

Figure 12 describes expansion of 8K on Figure 11.

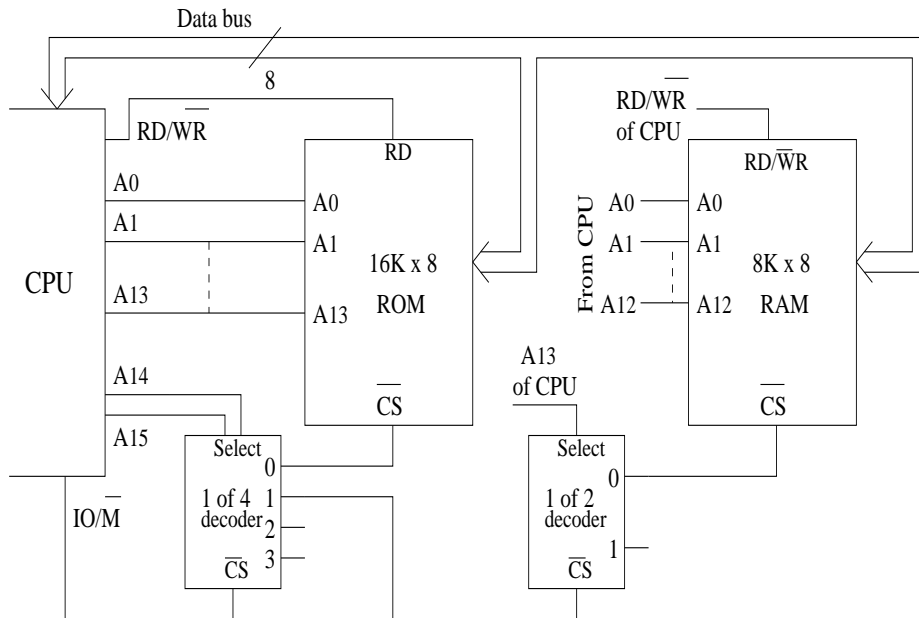


Figure 12: Memory expansion I

Address space occupied by the memory chips

Address of	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	\dots	A_1	A_0	Chip
LS location	0	0	0	0	0	\dots	0	0	0000h ROM
MS location	0	0	1	1	1	\dots	1	1	3FFFh
LS location	0	1	0	0	0	\dots	0	0	4000h RAM
MS location	0	1	0	1	1	\dots	1	1	5FFFh

-that is, ROM occupied from 0000h to 3FFFh and RAM 4000h to 5FFFh.

Expansion in complete decoding

Figure 13 describes expansion of 32K on Figure 11.

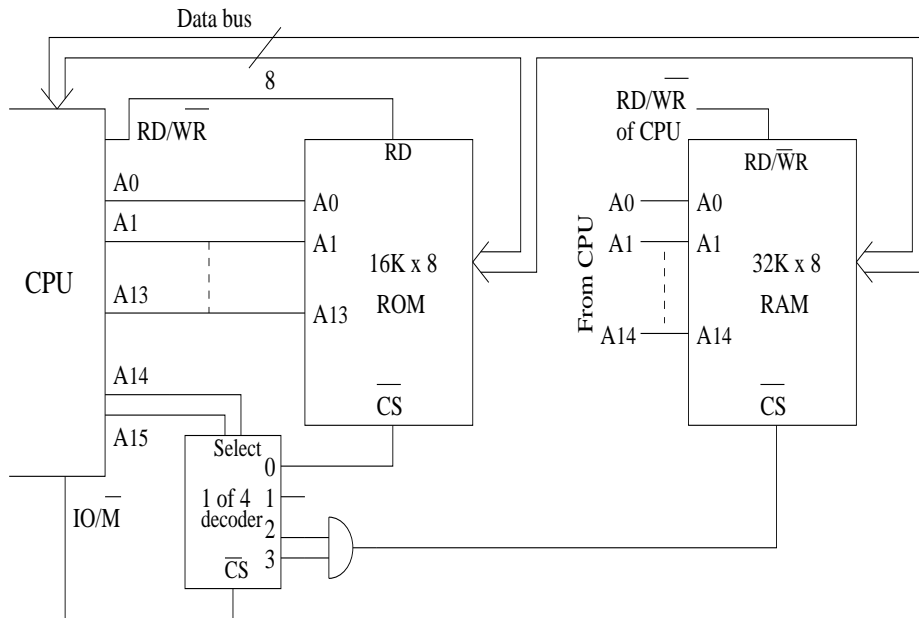


Figure 13: Memory expansion II

Address space occupied by the memory chips

<i>Address of</i>	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	\dots	A_1	A_0	<i>Chip</i>
<i>LS location</i>	0	0	0	0	0	\dots	0	0	<i>ROM</i>
<i>MS location</i>	0	0	1	1	1	\dots	1	1	
<i>LS location</i>	1	0	0	0	0	\dots	0	0	<i>RAM</i>
<i>MS location</i>	1	1	1	1	1	\dots	1	1	

-that is, ROM occupied from 0000h to 3FFFh and RAM 8000h to FFFFh.

0.2 High Speed Memories

Memory access time is a bottleneck.

A solution addresses optimal memory reference only through load and store instructions (implemented in RISC). Other solutions target

1. Decrease memory access time by using a faster but expensive technology.
2. Insert small faster memory (cache) in between conventional MM and CPU.
3. Access more than one word from memory in a cycle.
4. Interleave access to more than one word to achieve faster access.
5. Consider large memory word to retrieve more information in one cycle.

0.3 Main Memory Technology

1-D memory Each cell is connected to one address driver. If storage capacity is N bits, access circuitry needs one N -output decoder and N address drivers (Figure 14).

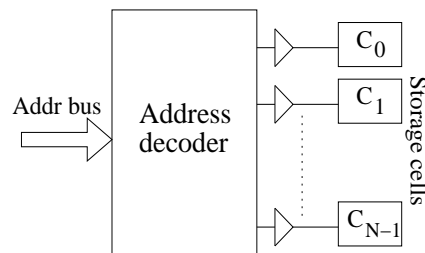


Figure 14: 1-dimensional memory module

2-D memory In 2-dimensional memory module, number of cells =

$$2^{k_1} \times 2^{k_2}, \text{ where } k_1 + k_2 = m.$$

Number of address drivers is $= 2^{k_1} + 2^{k_2}$ (Figure 15).

This can also be modified as

2^{k_1} rows $\times 2^{m+k_2}$ columns (2^m columns are selected by each one of 2^{k_2} columns).

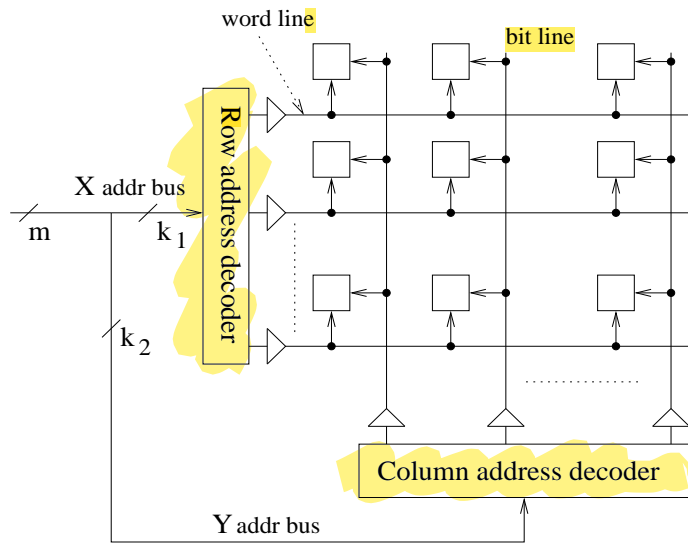


Figure 15: 2-dimensional memory module

3-D memory Home task.

0.3.1 Constructing large memory module

Objective is to solve memory design problem that a computer architect considers:

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

Memory with 2^m words each of d bits is called $2^m \times d$ -bit memory.

General approach - construct $2^p \times q$ array of $M_{2^m, d}$, where $p = \lceil m_1/m \rceil$ and $q = \lceil d_1/d \rceil$.

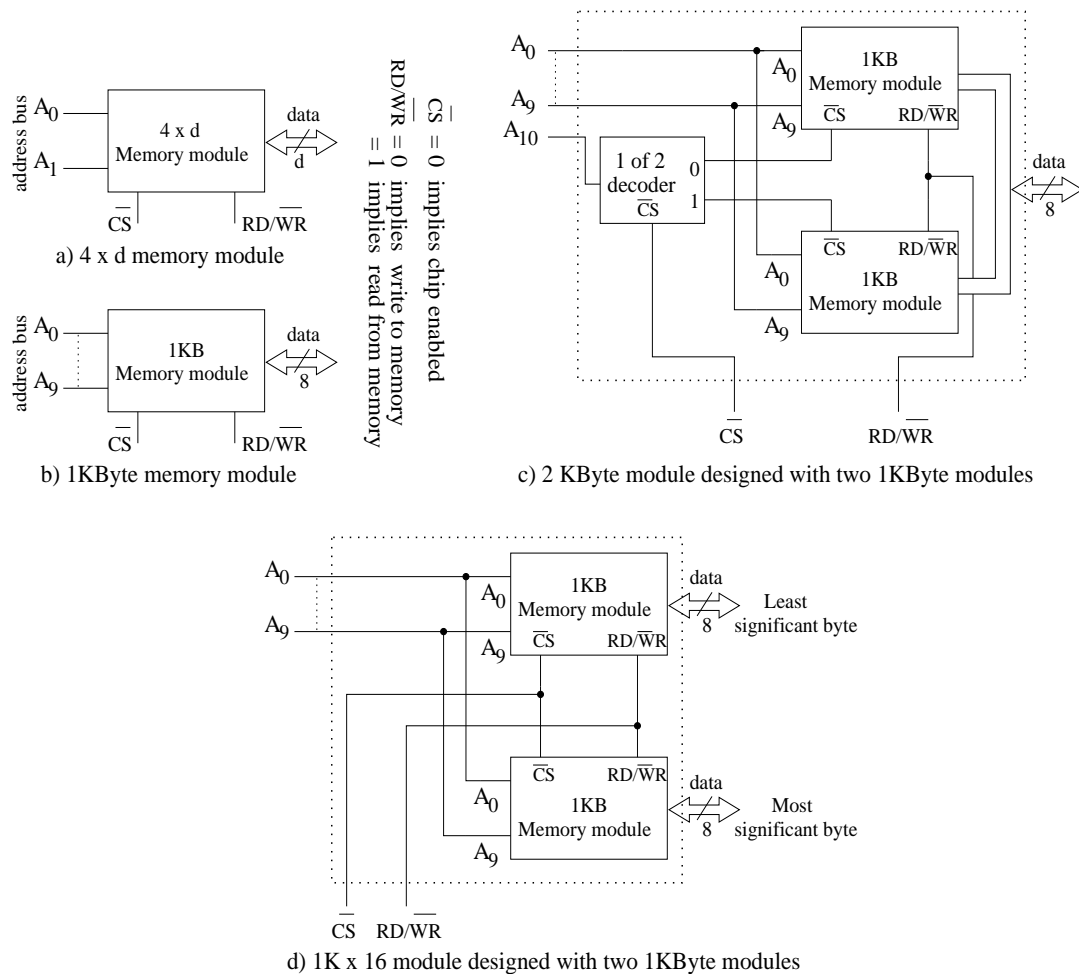


Figure 16: Memory arrays

QUIZ

Lecture 21: March 30, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

0.3.4 Content Addressable Memory (CAM)

Content addressable memory (CAM), also known as associative memory.

A CAM cell has storage capability as well as circuitry to realize matching its content with an argument. A CAM block is shown in Figure 23.

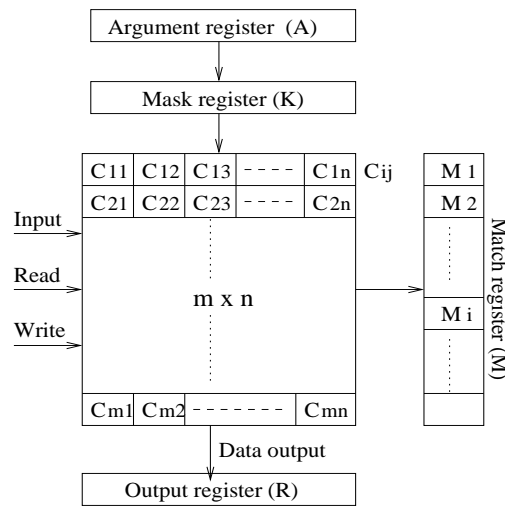


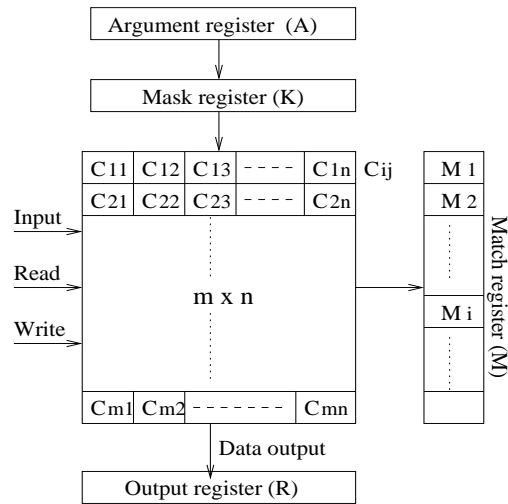
Figure 23: Content addressable memory (CAM)

Each word ($C_{i1}C_{i2}C_{i3} \dots C_{in}$) is compared in parallel with argument register A ($A_1A_2A_3 \dots A_n$) content.

Also gives due consideration of mask register K.

$K_j = 0 \Rightarrow j^{th}$ bit (C_{ij}) of word (i) is not to be compared.

If there is a match between A and word i , respective bit M_i of M is set to 1.



Let consider,

$$\begin{aligned}
 A &= 1010\ 1101\ 0011\ 1110 \\
 K &= 1111\ 0000\ 0000\ 0000 \\
 word1 &= 1100\ 1101\ 0011\ 1110 \\
 word2 &= 1010\ 0000\ 0000\ 1100
 \end{aligned}$$

Here, all K_1 , K_2 , K_3 and K_4 are 1.

That is, only four MSBs of A are to be compared with corresponding bits of $word_1$ / $word_2$.

For $word_1$, there is no match - and M_1 of M is set to 0.

For $word_2$, a match is found and M_2 is set to 1.

Match logic: Match logic is shown in Figure 24(b).

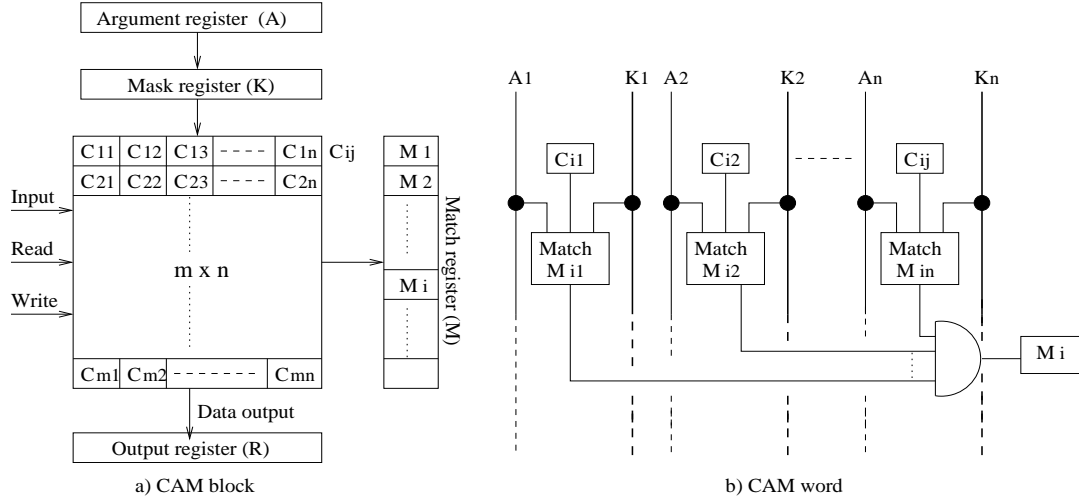


Figure 24: Content addressable memory (CAM)

Match logic M_{ij} compares j^{th} cell of i^{th} memory word with j^{th} bit (A_j) of A.

If all M_{ij} s, for an i are 1 -that is $\prod_j M_{ij} = 1$, then $M_i = 1$.

Word i of a CAM matches with argument A -that is, M_i is set if in Figure 24(b)

$$M_{ij} = A_j C_{ij} + A'_j C'_{ij} + K'_j \text{ is true for all } j = 1 \text{ to } n.$$

0.4 Cache Memory

Follow Figure 25.

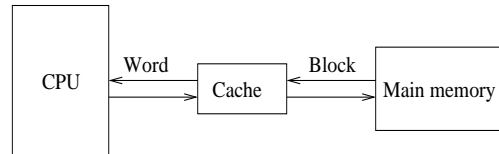


Figure 25: Cache main memory data transfer

Objective of the memory hierarchy (cache-MM) is to increase cache hit ratio (h).

$$h = \frac{N_1}{N_1 + N_2}$$

N_1 = number of times the word sought by the CPU is available in cache.

N_2 = number of times the word sought by CPU is not available (miss) in cache.

$$\text{miss ratio} = 1 - h.$$

miss penalty = time to transfer a block from MM to cache + time to deliver the item to CPU from cache.

Miss penalty is to be minimized.

Average access time is

$$\text{AMAT} = h \times t_{\text{cache}} + (1-h) \times t_2.$$

t_2 = time taken to fetch from main memory to CPU bypassing cache or through cache as per the system design and architecture.

Several functions guide the performance of a cache based system -

cache size, block size, associativity: mapping function, replacement policy.

Cache to MM write policy (write-back or write-through).

0.4.1 Mapping function

1. Direct mapping
2. Associative mapping
3. Set associative mapping (m -way associative mapping).

0.4.2 Direct mapping

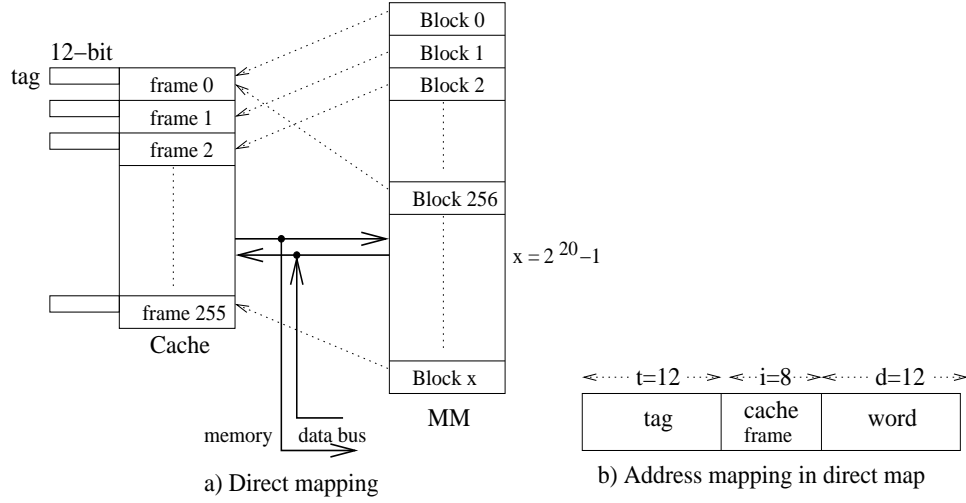


Figure 26: Direct mapping technique

Example: MM is of 2^{32} byte, page size = 4KB and cache is of 1MB (2^{20}).

Number of cache frames is

$$\frac{2^{20}}{2^{12}} = 2^8 = 256$$

Block i is mapped onto the frame (i modulo 256).

Number of blocks in MM is

$$\frac{2^{32}}{2^{12}} = 2^{20} = 1\text{M},$$

Therefore,

$$\frac{2^{20}}{2^8} = 2^{12} = 4\text{K}$$

MM blocks compete for a single cache frame.

That is, cache frame 0 can be occupied by MM blocks 0, 256, 512, \dots

12-bit tag is associated with each frame to signify currently residing block number.

Direct mapping is a many-to-one mapping.

0.4.3 Associative mapping

A main memory (MM) block can potentially reside in any cache frame.

So, if 1M main memory block is competing for 256 cache frames, 20 tag bits are required to identify an MM block in cache.

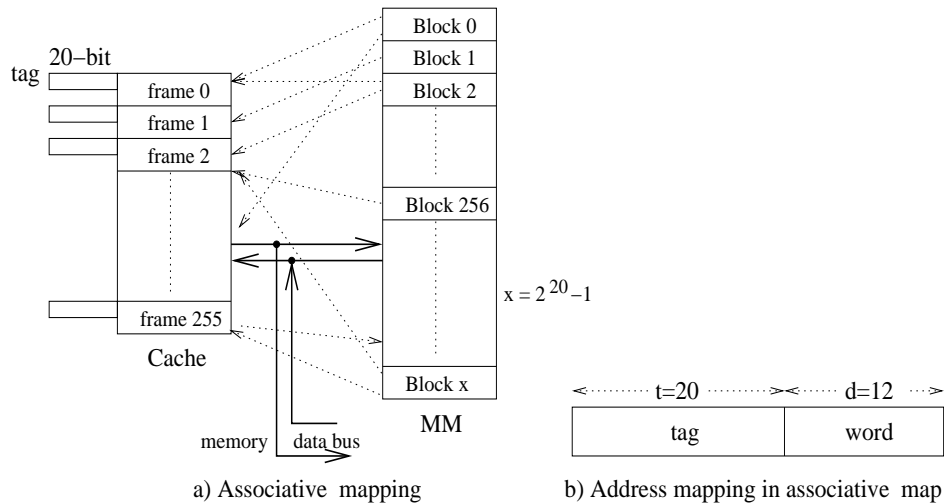


Figure 27: Associative mapping technique

Here, tags are stored in CAM so that all the tags can be compared simultaneously.

Lecture 22: April 6, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

0.4.4 m -way set-associative mapping

In direct mapping of Figure 26, 4K to 1 mapping is considered.

If we realize m -way associative mapping for it, then mapping is $m \times 4K$ to m .

In Figure 28, a 2-way set-associative mapping is shown.

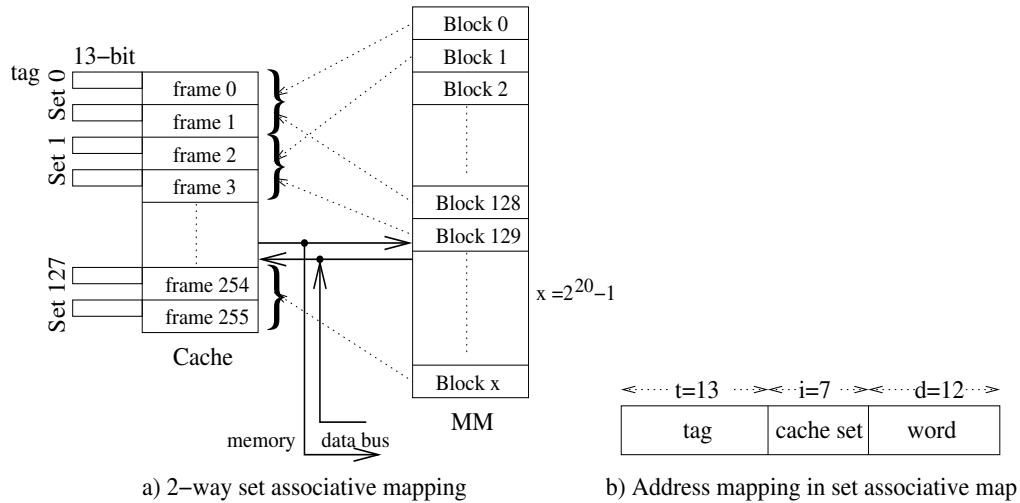


Figure 28: m -way set associative mapping technique

Here, number sets is $= 128 = 2^7$.

That is,

$\frac{2^{20}}{2^7} = 2^{13}$ MM blocks are competing for a set. This implies tag $t = 13$ (Figure 28).

The set is also called *index* field.

If m is 4 to 8, its efficiency is almost equivalent as associative mapping.

Direct mapping is an 1-way set associative mapping mapping.

To speed up, in Figure 29, index is compared with two tags of a set.

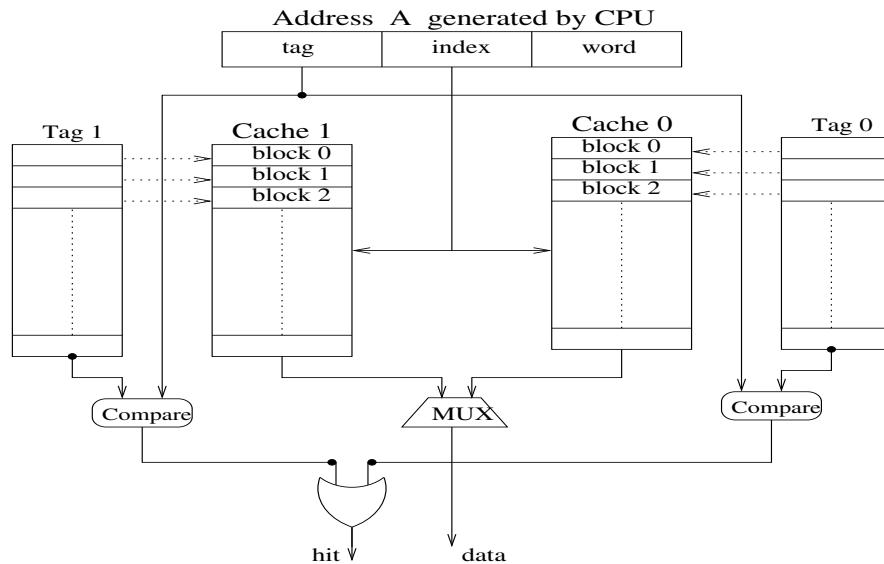


Figure 29: Architecture implementing 2-way set associative mapping scheme

Comparison of mapping techniques is shown in Figure 30.

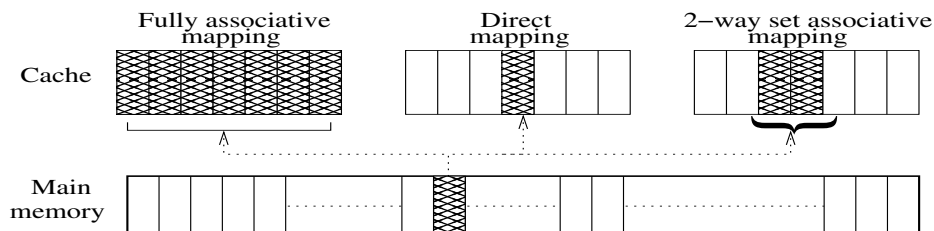


Figure 30: Comparison of mapping techniques

0.4.5 Replacement algorithms

Cache system implementing direct mapping does not require replacement policy. There is no other option but replace the block in cache frame selected for placement. For set associative and fully associative mapping, we need replacement policy.

Three algorithms are considered for such replacement of blocks:

(i) Random replacement,

(ii) FIFO, and

(iii) LRU (least recently used block is replaced) or an approximation.

0.4.6 Cache write policies

It determines when an update to a cache word is forwarded to the main memory.

Two cache write policies - write-back and write-through.

a) *write-back*: Update to block B in cache is written in MM only when replacing B from cache.

It avoids miss penalty for every write operation.

b) *write-through*: While writing B at cache, similar write operation is performed in MM. That is, each write operation in cache is the write miss

The blocks in MM are the latest updated blocks.

Write through is very effective in multi-processor system.

However, successive operations will result in a bottleneck.

To exploit efficiency of write-through, faster implementation of write-through is done with introduction of a buffer in between cache and MM (Figure 31).

Typically, a write buffer is of 4/5 blocks.

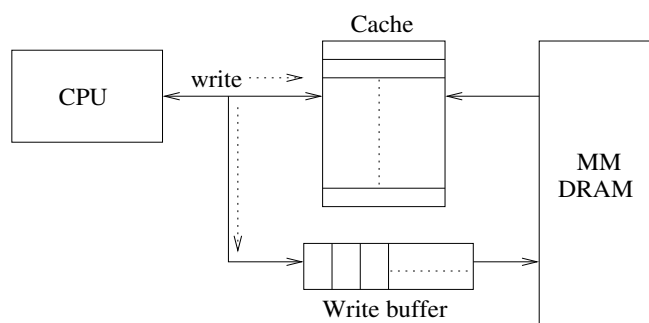


Figure 31: Write buffer

0.5 PRAM

CPU can float addresses much faster than settling time of a memory cell.

For multiple access, CPU generates $addr_1, addr_2, \dots, addr_i, addr_{i+1}, \dots$.

There is single decoding logic - overlapping of requests is difficult to realize.

PRAM (parallel RAM) allows access to more than one locations in a cycle.

Ideal PRAM is not realized. Different organizations of PRAM are proposed.

0.6 Interleaved Memories

Interleaved memories allow simultaneous access from from main memory.

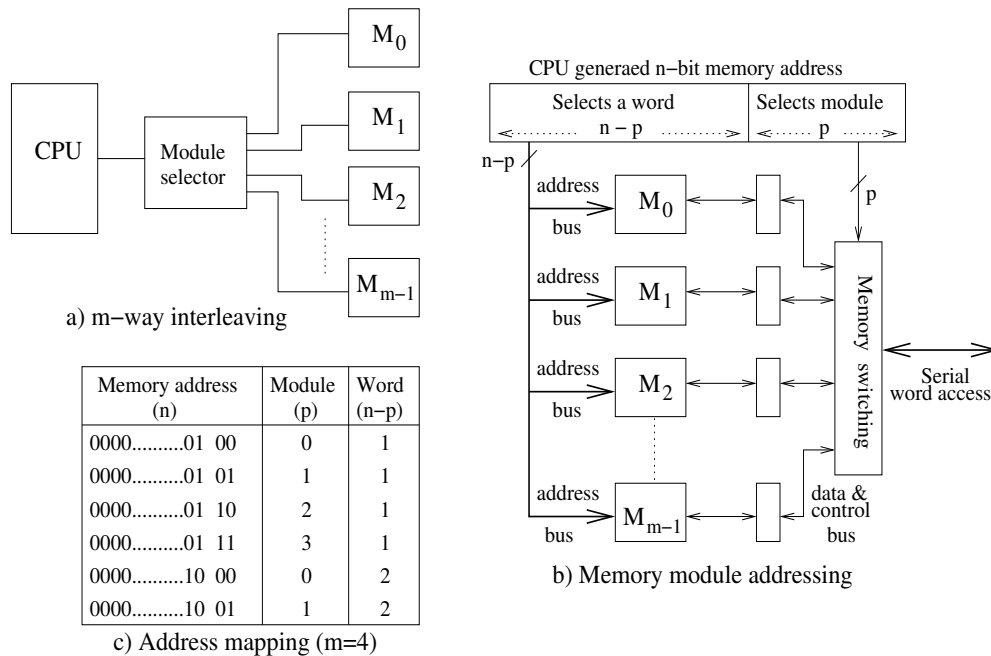


Figure 32: Memory interleaving

Memory is partitioned into m separate modules M_0, M_1, \dots, M_{m-1} (Figure 32(a)).

Each module is provided with its own addressing circuitry.

Consecutive memory references A_i and A_{i+1} fall on to different modules (M_j/M_{j+1}).

For m modules, it is m -way interleaving.

0.7 Large Memory Word

Large word size was considered at the early phase of our modern computer.

The recent development is the VLIW (very large instruction word).

In VLIW, in a cycle, a long memory word is fetched.

Performance of Memory

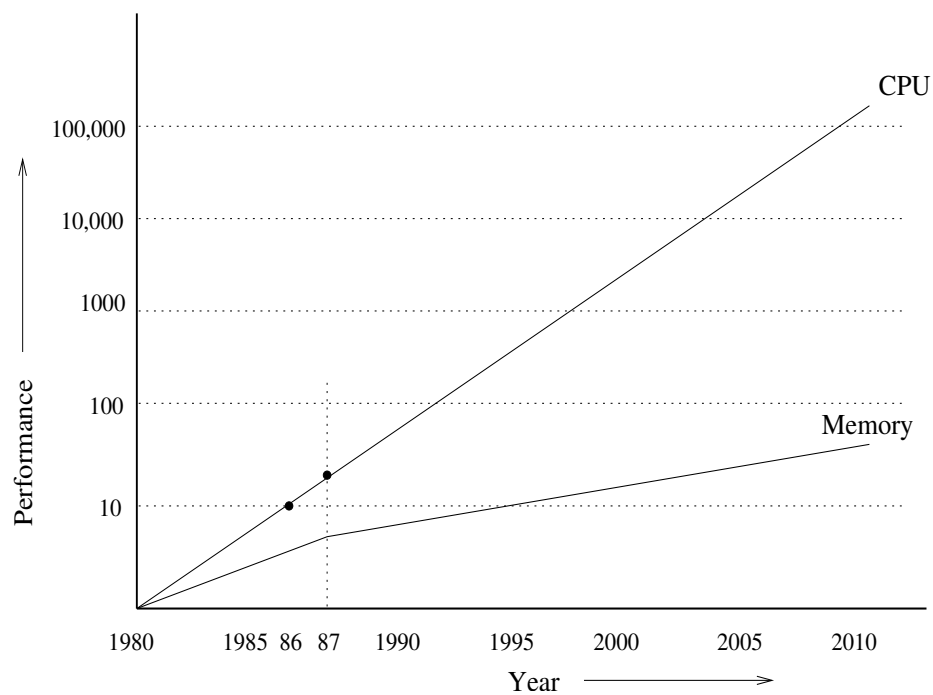


Figure 33: Memory performance

A memory hierarchy is a way of organizing computer memory in a hierarchical order based on the speed, capacity, and cost of the different types of memory. In general, the closer a memory type is to the CPU, the faster it is and the smaller its capacity and higher its cost per bit.

Memory hierarchy

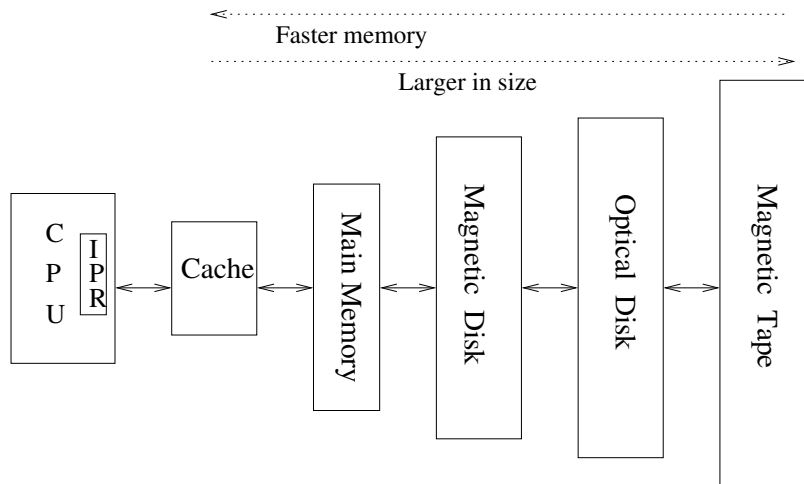


Figure 34: Memory hierarchy

Virtual Memory

Virtual memory (VM) system gives programmer an illusion that he/she has a very large memory at his/her disposal, even though m/c has a relatively small MM.

For example, let a CPU with 32 address lines is having 1MB actual MM. The virtual address space of the CPU is 2^{32} .

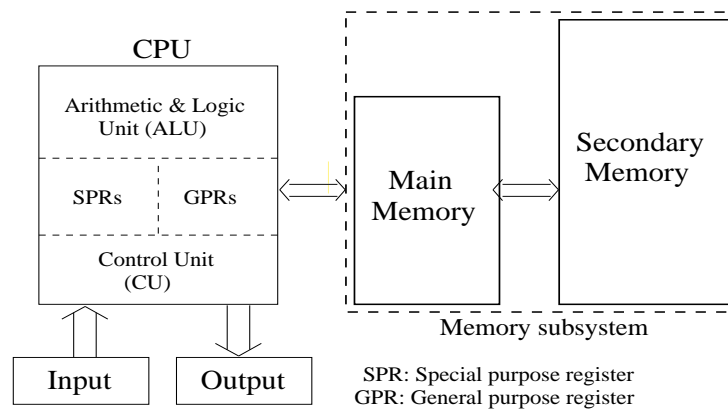
A mechanism is needed to map 32-bit VM address to a physical address of 20-bit.

Lecture 23-24: April 9, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

Peripherals



A machine computer without an input/output device is of no use.

Conventionally, CPU data processing speed is much much faster than input to CPU/CPU to output data transfer rate.

0.1 Properties

External I/O device normally communicates with I/O controllers interfaced with CPU through system bus (also used for data transfer between CPU and memory).

A system bus is shown in in Figure 1.

Command signals denote operations to be performed such as *memory read*, *memory write*, *I/O read*, *I/O write*, *transfer ACK*¹, *bus request*², *bus grant*³, *interrupt request*⁴, *interrupt ACK*⁵, *clock*, *reset*, etc.

Performance of a bus structure is measured by defining its *bandwidth*.

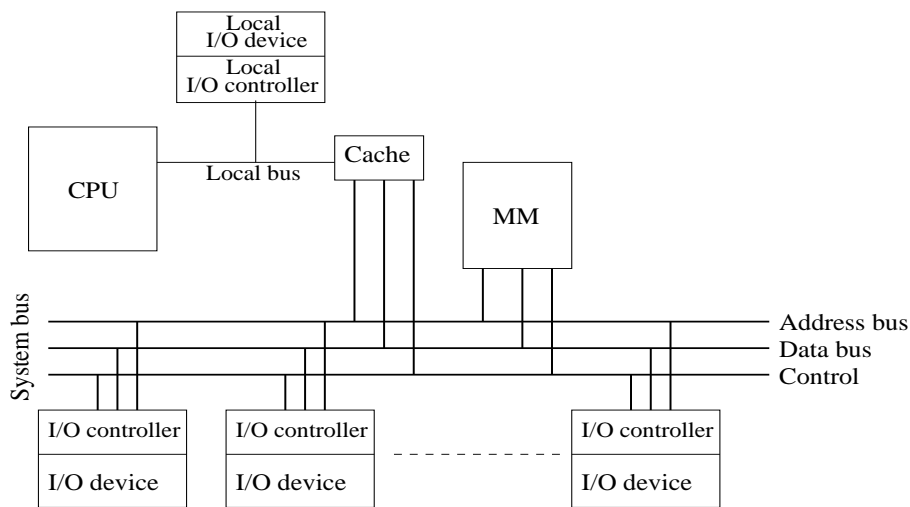


Figure 1: System bus

Such single-bus architecture provides uniform interfacing for devices (Figure 1).

It increases probability of bottleneck during data transfer.

¹Indicates that data have been accepted from or placed on the bus.

²Indicates that a module needs to gain control of the bus.

³Indicates that a requesting module has been granted control of the bus.

⁴Indicates that an interrupt is pending.

⁵Acknowledges that the pending interrupt has been recognized.

0.1.1 Multiple-bus structure

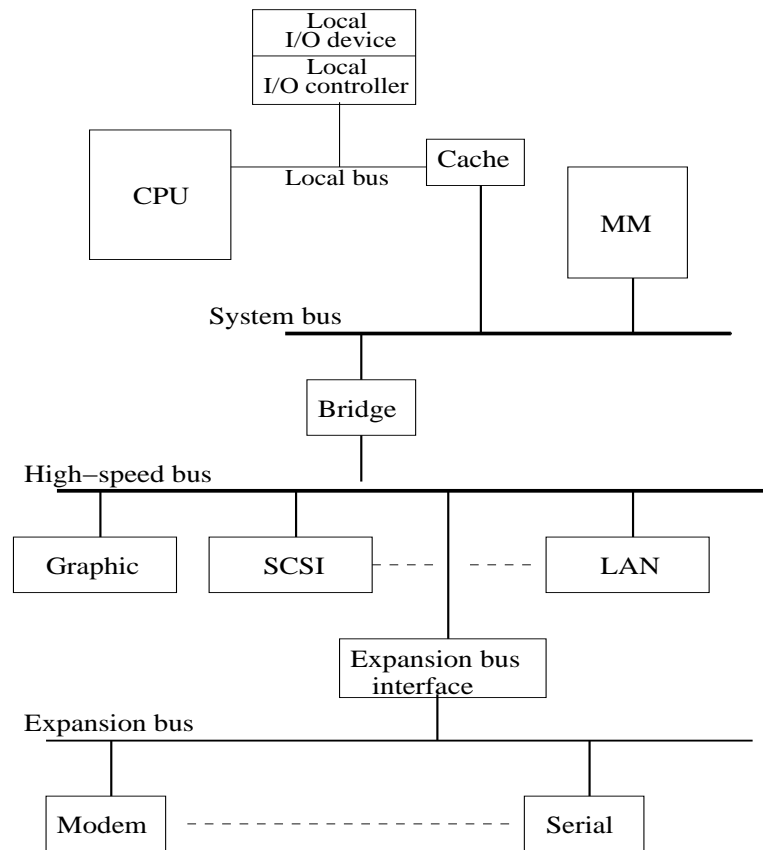


Figure 2: Multiple-bus structure

Multiple-bus structure (Figure 2) improves overall bandwidth.

It insulates memory-to-CPU traffic from I/O traffic.

SCSI (small computer system interface) supports disk drives and other peripherals.

The serial port is used to support printer/scanner.

Limitation : complexity of hardware and power consumption increases

Peripheral component interconnect (PCI), from Intel, is processor-independent bus.

PCI and PCI Express (PCIe) deliver better system performance for high-speed I/O.

0.1.2 Problems with the I/o interfacing

Reasons for which the peripherals are not directly connected to system bus are:

1. Wide variation of I/O devices - of type mechanical, electrical, electromechanical and electronic. Even transducers, ADCs or DACs are used.
2. Speed variations
3. Wide variation in format of data : serial, parallel, ...

0.1.3 Data transfer schemes

Data transfer between micro-processor and I/O, and memory and I/O devices.

Timing incompatibility among processor and I-O devices may cause problems.

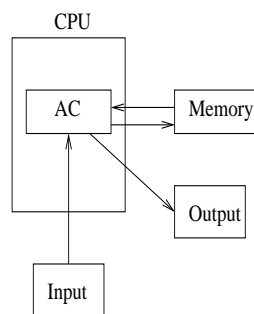


Figure 3: Programmed I/O

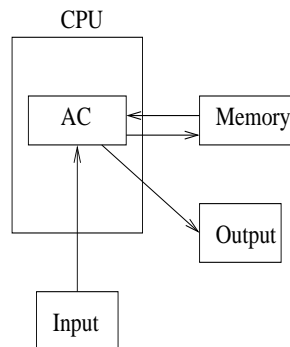
- i) Programmed I/O
- ii) DMA (direct memory access).

DMA bypasses CPU.

0.2 Programmed I/O Data Transfer

CPU executes program - it results in data transfer (between I/O device and memory).

Completely controlled by CPU and it is (i) from input to CPU (say, AC) and from CPU to memory; or (ii) from memory to CPU and then from CPU to output device.



Three alternative schemes in programmed I/O -

1. Synchronous (mode) data transfer
2. Asynchronous (mode) data transfer
3. Interrupt driven (mode) data transfer

0.2.1 Synchronous mode of data transfer

CPU is aware of speed of I/O device.

CPU synchronizes its own speed with that of I/O device speed.

CPU and I/O device share a common clock.

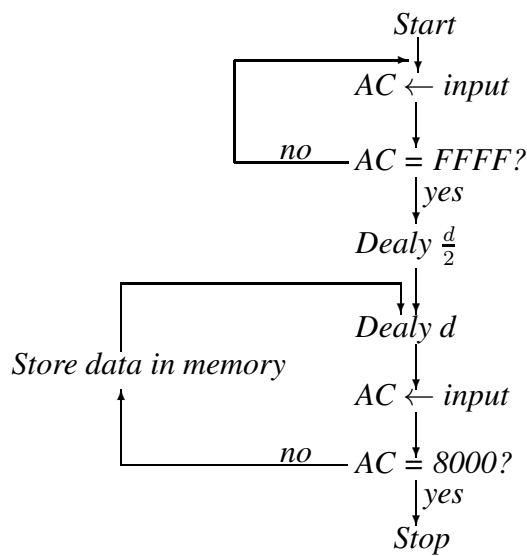
Example 0.1 A device transmits 16-bit data packet with start of transmission as all "1"s code and 8000_H at end of transmission (packet).

Device prepares a data at a delay of d time unit.

Synchronous data transfer scheme is shown in Algorithm 0.1.

Delay of $\frac{d}{2}ms$ in Algorithm 0.1 is to get steady state data.

Algorithm 0.1 *Synchronous data transfer*



May be used when I/O device and a processor match in speed.

0.2.2 Asynchronous/Handshaking mode of data transfer

Is effective when processor/CPU does not know the speed of I/O device.

Suitable for low-speed I/O devices.

An unit can use its own private clock.

CPU checks whether input [output] device is ready to send [accept] next data.

Follow Figure 4.

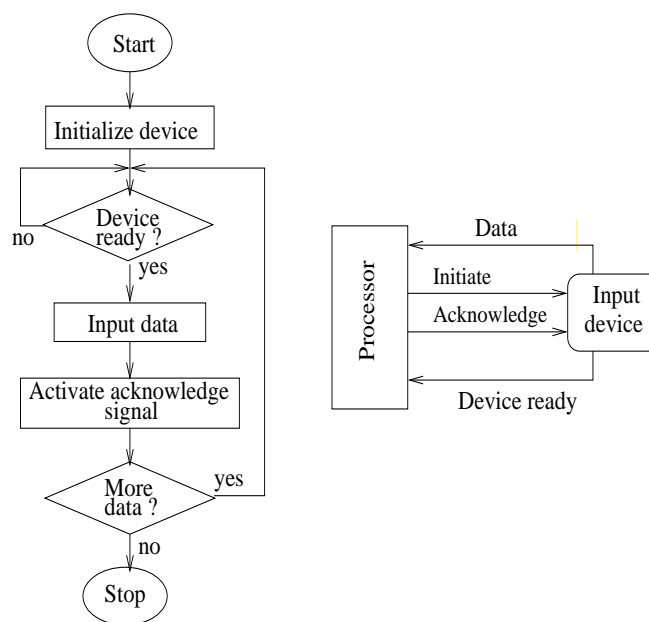


Figure 4: Asynchronous data transfer

Step 1. When input device is ready to transfer new data it outputs the data on data bus and sends a data ready signal

Step 2. CPU continuously tests status of data ready signal until it is active. When active CPU reads in data from data bus, and sends acknowledgement signal

Step 3. When input device receives acknowledgement signal, it deactivates data ready line, and goes back to Step 1

Example 0.2 Data transfer from analog to digital converter to processor (Figure 5).

SOC: start of conversion, EOC: end of conversion of analog input (V_A).

a pulse is computed in this line when conversion is complete.

CPU always checks EOC to read data.

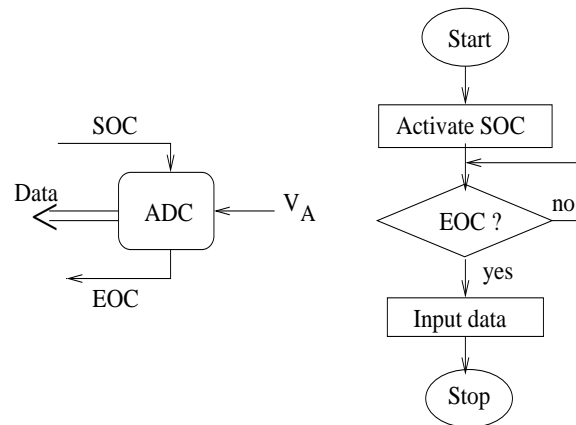


Figure 5: Asynchronous data transfer from ADC to CPU

CPU remains busy to get next data -that is, CPU waste time and can't do other job.

0.2.3 Interrupt driven data transfer

CPU can proceed to execute its job/program.

For data transfer from input device to CPU, input device issues an interrupt request.

When CPU receives device ready signal through an interrupt it process I/O transfer routine and then returns to the task it was originally performing.

Summary of the steps to be executed is given below.

Step 1. CPU initializes I/O devices (A/D converter).

Step 2. CPU continues excution of subsequent instructioni in the program.

Step 3. I/O device sends signal (interrupt) to CPU when it is ready.

Step 4. CPU completes execution of current instruction and grants the interrupt.

(a) It saves PC on a stack.

(b) Branches to location IBA (Int. Branch Add) where ILS, ISS are stored.

(c) CPU executes ISS, saves processor registers, stack pointer etc on stack.

(d) CPU then transfers data.

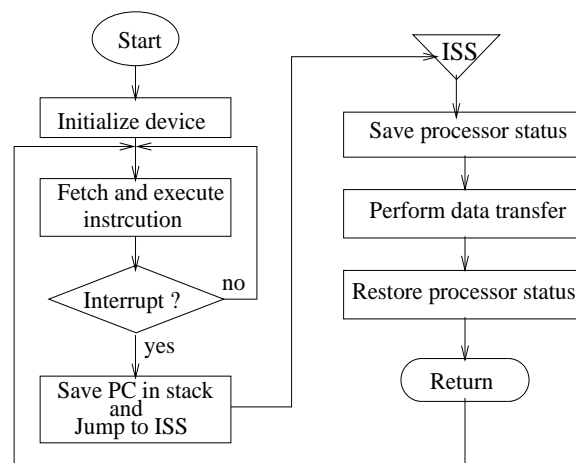


Figure 6: Interrupt driven data transfer

This scheme is effective for relatively slow I/O devices (keyboard/ADC/DAC/printer/etc).

0.3 DMA

An external device - DMA controller (DMAC), on behalf of CPU, transfers data directly from input device to memory as well as memory to output device.

DMA transfer takes place without execution of IO instructions by CPU.

Prior to a data transfer, CPU has to initialize DMA controller with:

1. Length of the block to be transferred,
2. Starting address of memory (to) from which data transfer should occur.

During the DMA data transfer the CPU is forced to "hold on".

CPU suspends its current activities and attends DMA request.

CPU can respond to a DMA request after current m/cycle (Figure 8).

CPU releases system bus and signals by activating DMA acknowledgement signal.

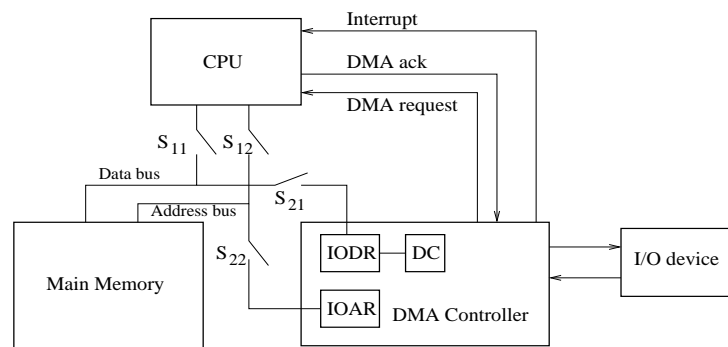


Figure 7: DMA interfacing

DMA break point refers to a mechanism used by DMA controllers to pause or halt a DMA transfer in progress.

A DMA transfer may need to be paused or halted in certain situations, such as when a memory location that the DMA controller is trying to access is not available or is being used by another device. The DMA break point allows the DMA controller to detect such situations and halt the DMA transfer temporarily.

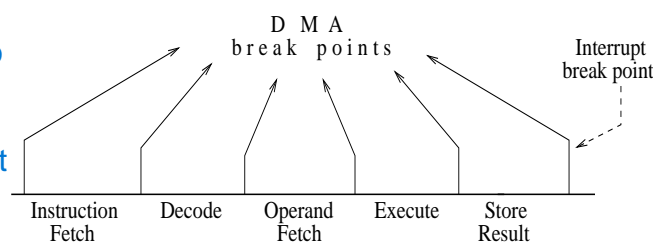


Figure 8: DMA break points

When a DMA break point is reached, the DMA controller temporarily suspends the transfer and waits for the situation to be resolved. The CPU can then intervene and take the necessary action to resolve the issue, such as freeing up the required memory location or notifying the other device to release the memory.

they help prevent data corruption or loss due to conflicts between different devices accessing memory simultaneously.

Lecture 25: April 13, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

0.3.1 DMA transfer steps

Data counter (DC) of Figure 7 contains number of words to be transferred.

IO addr register (IOAR) contains base address of MM considered for data transfer.

Input output data register (IODR) temporarily stores data that is to be transferred.

Following steps realize a DMA data transfer from an input device to MM.

Step 1. CPU initializes the DC and IOAR

Step 2. DMAC checks status of Input device. If device is ready to transmit data, DMAC activates DMA request (DMAR).

Step 3. As soon as CPU senses DMAR, CPU waits for next DMA breakpoint. It then relinquishes control of buses and activates DMA acknowledge (DMAack).

Step 4. Upon receiving DMAack, DMAC transfers data directly to MM.

Step 5. After a word is transferred, IOAR and DC are updated.

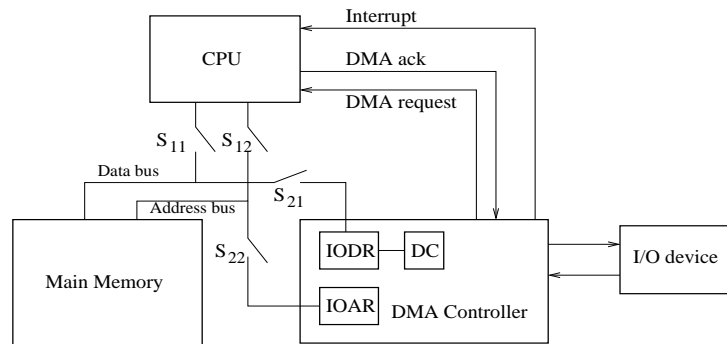
$$\text{IOAR} \leftarrow \text{IOAR} + 1$$

$$\text{DC} \leftarrow \text{DC} - 1$$

If $\text{DC} > 0$, then go to step 2

Step 6. If $\text{DC}=0$, then DMAC relinquishes control of system bus, and sends an interrupt to signal end of data transfer.

DMA data transfer from main memory to output device follows similar process.



Two types of DMA data transfer techniques are used in microprocessors:

1. Microprocessor halt DMA. It either implements

- a. Block transfer or
- b. Cycle stealing

while stopping the processor.

2. Interleaved DMA - without stopping the processor.

Interleaved DMA: each data transfer includes one byte per instruction cycle.

Example: when processor is decoding instruction or performing ALU operations, DMAC can transfer a byte.

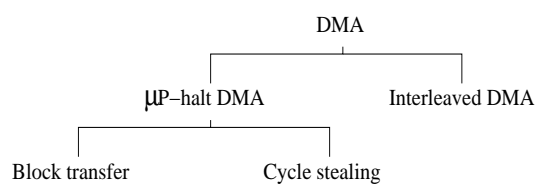


Figure 9: DMA classification

0.3.2 Bulk DMA

CPU is suspended for entire duration for extremely fast I/O device.

It is needed by secondary memory devices where data transfer cannot be stopped.

With block DMA, maximum IO data transmission rates can be achieved.

It may require the CPU to remain inactive for comparatively long periods.

0.3.3 Cycle stealing

Bus cycles are stolen only when CPU is not using system bus.

CPU is operated by an external clock - this is accomplished by not providing clock signal to CPU. An INHIBIT signal is used for this purpose.

QUIZ

Lecture 26-27: April 23, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

Pipelining

Parallel processing is a solution to achieve high speed computation.

Number of parallel processing mechanisms have been developed in uniprocessor.

Multiplicity of functional units (CDC 6600 had 10 functional units), IBM-360/91 (1968) having 2 execution units (one for fixed-point arithmetic and another for floating point arithmetic); overlapped CPU and I/O operations (I/O processors, DMA etc); use of hierarchical memory system; multiprogramming and time sharing etc.

Most significant attempt to achieve speed-up in computation is *pipelining*.

Pipeline breaks a process into N steps for N-fold increase in processing speed.

Figure 1 represents a simple linear pipeline.

Pipelines is extended to various structures of interconnected processing elements.

Arithmetic pipelining is used in some specialized computers.

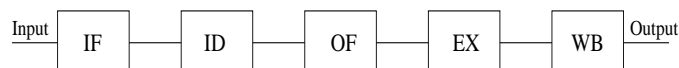


Figure 1: Instruction pipeline

Three categories - instruction pipeline, arithmetic pipeline and processor pipeline.

0.1 Instruction Pipeline

Instruction pipeline: various phases - instruction fetch (IF), decode (ID), operand fetch (OF), arithmetic and logic execution (EX), and store result (WB).

Figure 1 describes such a 5-stage instruction pipeline.



1. Instruction fetch (IF) Processor reads next instruction to be executed.
2. Instruction decode (ID) Processor works out what this instruction is.
3. Operand fetch (OF) Processor reads values required by the operation.
4. Instruction execution (EX) Processor executes instruction on operand values.
5. Result storage (WB) Processor stores result of operation in register/memory.

Execution of 4 instructions in 5-stage pipeline is shown in Figure 2.

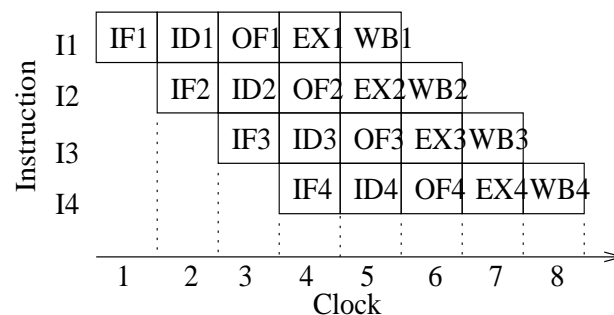


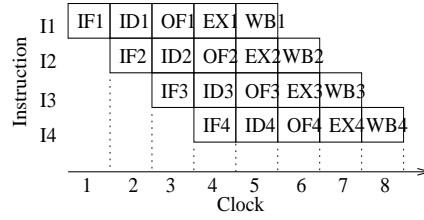
Figure 2: Instruction execution in 5-stage pipeline

0.1.1 Speed up

Execution time of n instructions, each passes through k stages, (without pipeline)

$$CPUtime_{non-pipelined} = n \times k \times CT$$

where CT is clock time taken for computation in each stage.



In Figure 2, first instruction I-1 get executed after 5 clock cycles.

After 6th clock I₂, after 7th I₃ and so on are completed.

That is, for k-stage pipeline, time taken for execution of n instructions is

$$CPUtime_{pipelined} = k \times CT + (n - 1) \times CT .$$

So speed-up of the k -stage pipeline execution is

$$\begin{aligned}
 Speedup_k &= \frac{CPUtime_{non-pipelined}}{CPUtime_{pipelined}} \\
 &= \frac{n \times k \times CT}{k \times CT + (n - 1) \times CT} \\
 &= \frac{n \times k}{k + n - 1} \\
 &= \frac{k}{\frac{k}{n} + 1 - \frac{1}{n}} \\
 &\simeq k
 \end{aligned}$$

as for a program $k \ll n$, the factors $\frac{k}{n}$ and $\frac{1}{n}$ are close to 0.

In case of 5-stage pipeline of Figure 1 and $n = 10,000$, the actual speed-up

$$Speedup_5 = \frac{10000 \times 5 \times CT}{(5 + 10000 - 1) \times CT} = \frac{50000}{10004} = 4.998 \simeq 5.$$

Actual speed-up is limited due to other factors - stages need not have same delay i.e, CT in non-pipelined architecture may differ from CT in pipelined architecture.

0.1.2 Latency/CPI

Pipelined execution of Figure 3 assumes latency 1.

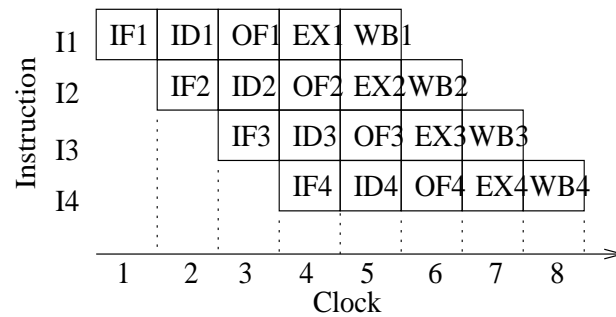


Figure 3:

Delay between issuing of two consecutive pipeline items is one pipeline cycle/clock.

CPI of a pipeline system is proportional to its latency.

CPI of instruction pipeline corresponding to Figure 3 is 1.

Pipelined execution shown in Figure 4 has latency = 2.

Delay between two consecutive pipeline items is two pipeline cycles. Its CPI is 2.

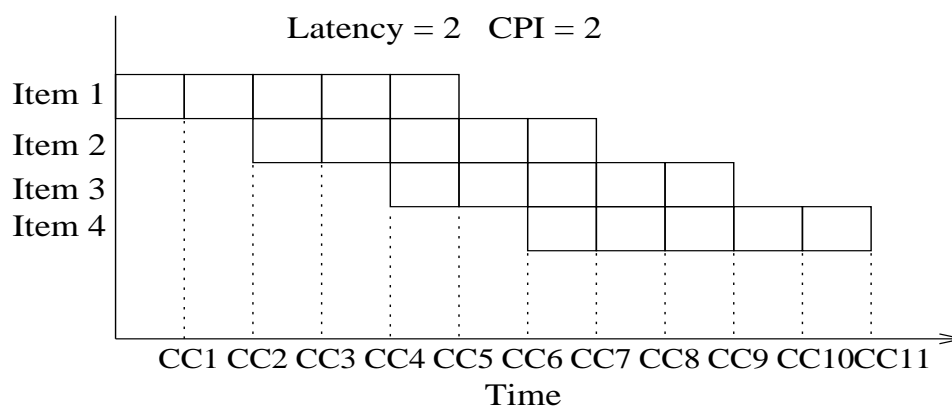


Figure 4: Pipelined execution with latency 2

0.1.3 Design issues

Several difficulties prevent instruction pipelining and hinder speed-up.

Construction of modules that run independently

Timing variations Not all the pipeline stages take the same delay.

The speed-up will be determined by the slowest stage.

Different instructions can have different operand requirements.

For significant timing differences in stages, following solutions can be considered.

Solution 1 Figure 5.

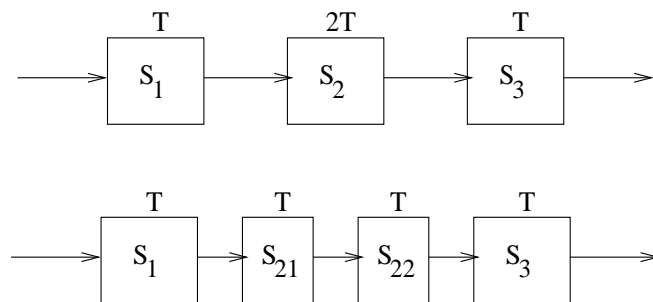


Figure 5: Timing mismatch : solution 1

However, independent moduling (S_{21}/S_{22}) may not be possible.

Another example

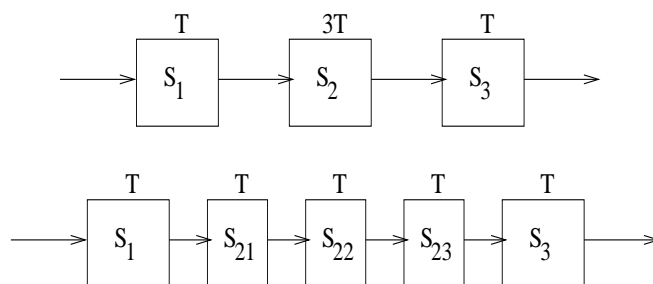


Figure 6: Timing mismatch : solution 1

Solution 2 Figure 7.

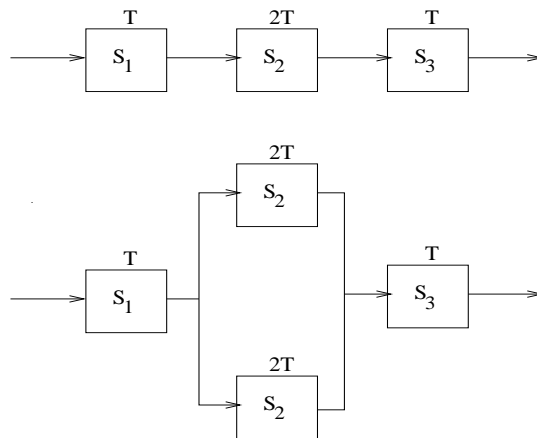


Figure 7: Timing mismatch : solution 2

Needs mechanism to divert pipeline items - one-to-many and many-to-one function.
Require MUX/deMUX and slow down processor. Speedup may not be close to k .

Another example

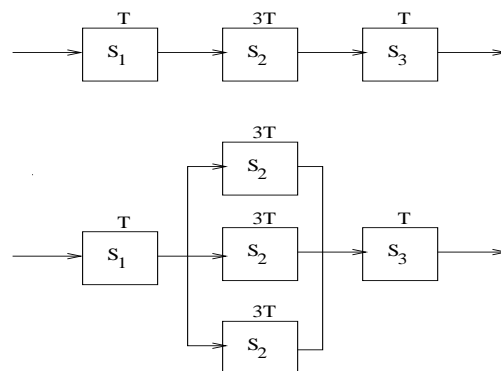


Figure 8: Timing mismatch : solution 2

Pipeline can be synchronous or asynchronous.

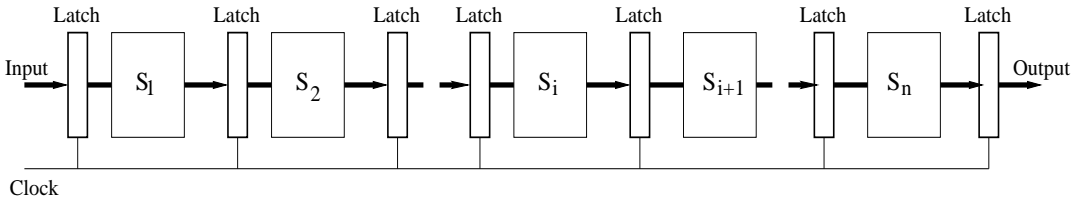


Figure 9: Synchronous pipeline

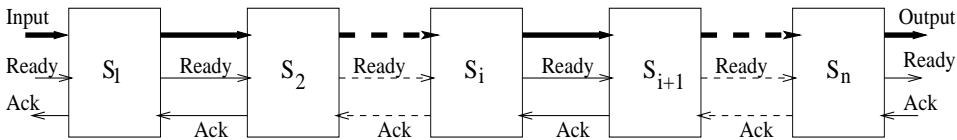


Figure 10: Asynchronous pipeline

Hazards

Structural hazard

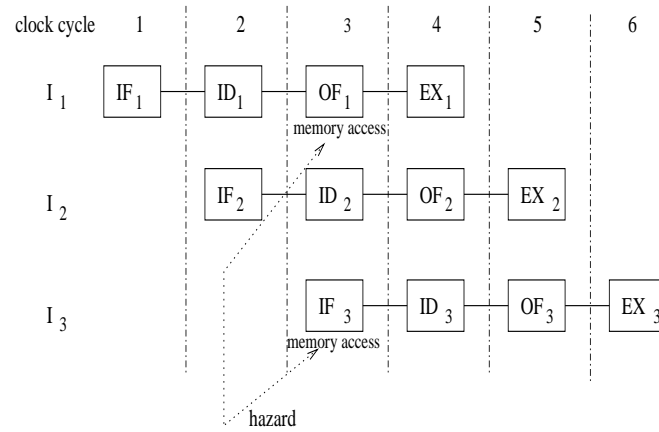


Figure 11: Structural hazard in 4-stage instruction pipeline

Data hazard

I_{i+k} must not be permitted to fetch an operand that is yet to be stored into by instruction I_i .

ADD R1, R2, R3
 MULT R4, R1, R5
 OR R6, R1, R7

Instructions MULT and OR, after ADD, use the result generated by ADD.

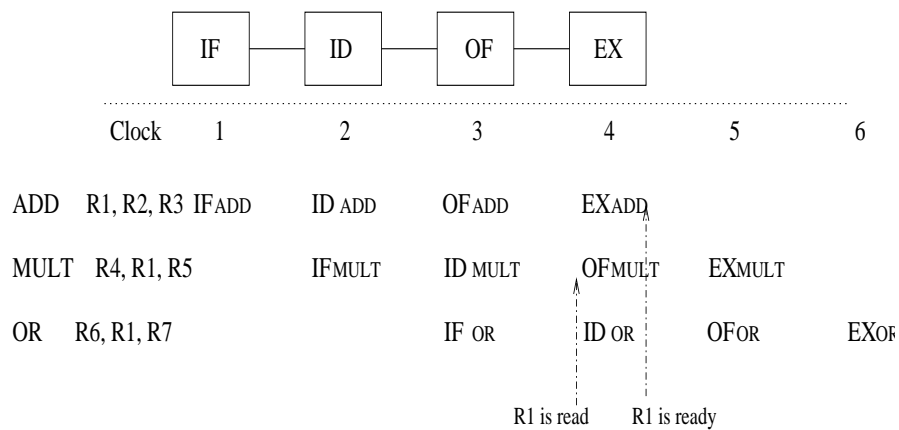


Figure 12: Data hazard in 4-stage instruction pipe

Control hazard

In order to fetch 'next' instruction, pipeline system must know which instruction is required next.

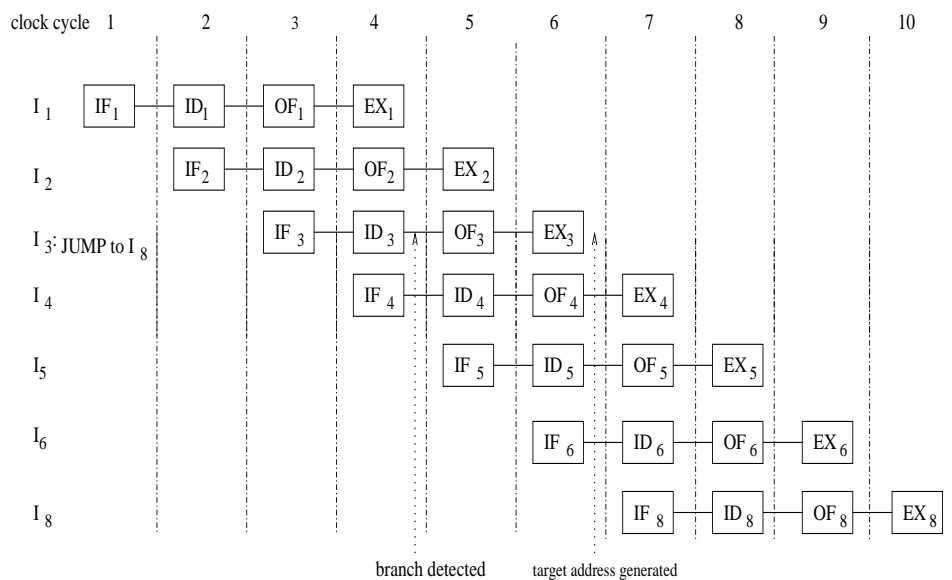


Figure 13: Control hazard in 4-stage instruction pipeline

If present instruction is a conditional branch, next instruction may not be known until current one is processed.

Pipeline may be slowed down by a branch instruction because we do not know which branch to follow.

Interrupts

Interrupts (Figure 14) insert 'extra' instructions into execution of instruction stream.

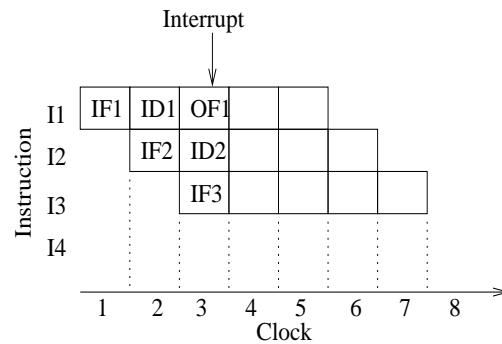


Figure 14: Interrupt in pipelining

Interrupt effect is taken when one instruction has completed.

But in pipelining, next instruction enters before current one has completed.

Solution 1: Simplest solution is to wait until all instructions in pipeline complete -that is, flush pipeline from starting point, before acknowledging interrupt.

For frequent interrupts, it makes pipeline inefficient; interrupt handling is delayed.

Solution 2: Select a point in the pipeline (say, EX stage). Instructions which have passed this point are only allowed to complete and then interrupt is acknowledged.

Lecture 28: April 27, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

0.2 Arithmetic Pipeline

Floating point addition/multiplication/division can be divided into number of phases.

Example pipeline for fixed point multiplication:

Multiplication logic used here is partial product logic.

Figure 15 is the 6-bit multiplier. It produces 6 (12-bit) partial products.

Each partial product is shifted left. The i^{th} partial product is shifted left i times.

Carry save adders, shift units and carry look ahead adder are used in a pipeline.

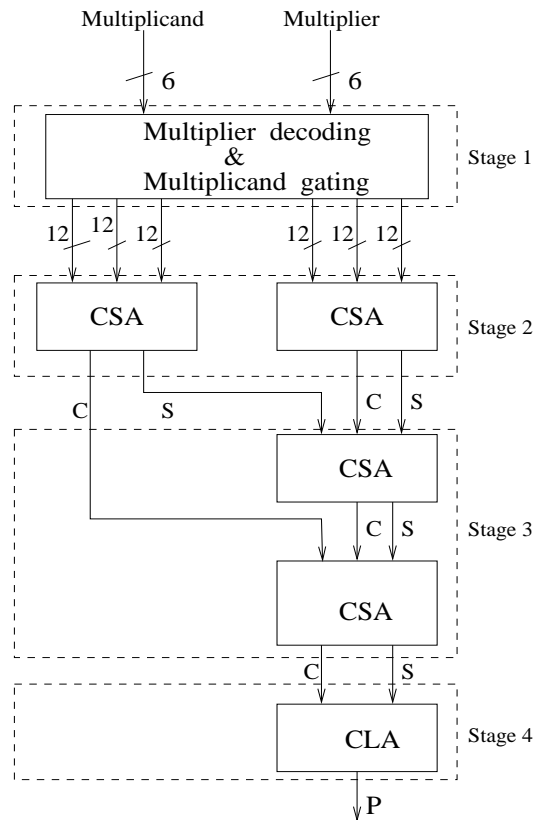


Figure 15: Arithmetic pipelining

Pipeline hardware for multiplication contains 4 stages.

Stage 1: contains logic for generating 6 partial products with appropriate shift.

Stage2 receives 6 partial products and generates 4.

Stage3 converts this into 2 partial products.

Stage 4: has a CPA, which finally generate product.

Here, the total gate delays in each stage are not same.

A design with almost same delay in pipeline stages is desirable.

A better design is shown in Figure 16.

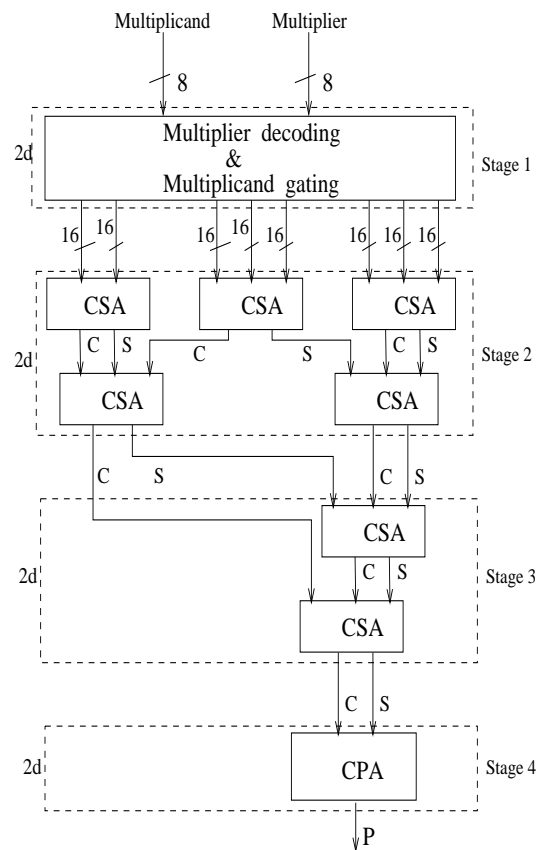


Figure 16: Arithmetic pipelining - 8-bit multiplication

0.3 Processor Pipeline

In processor pipelining, each stage of pipeline is a processing element (PE).

It performs different operations on same data stream by a cascade of processors.

This class of architecture is the multiple instruction single data (MISD) machine.

0.4 Linear Pipeline

Processing stages are cascaded linearly (Figure 17).

No feedback or feedforward path among pipeline stages.

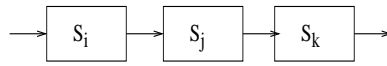


Figure 17: Linear pipeline

Asynchronous and synchronous models:

Delay of a pipeline stage S_i may not be same as that of stage S_{i+1} .

If delay of S_i is less compared to S_{i+1} , then S_i must hold its generated output so that S_{i+1} gets ready to accept this as input.

Data flow from stage S_i to S_{i+1} can be synchronized with a clock - synchronous pipeline (Figure 18).

Latches are introduced to handle delay mismatch.

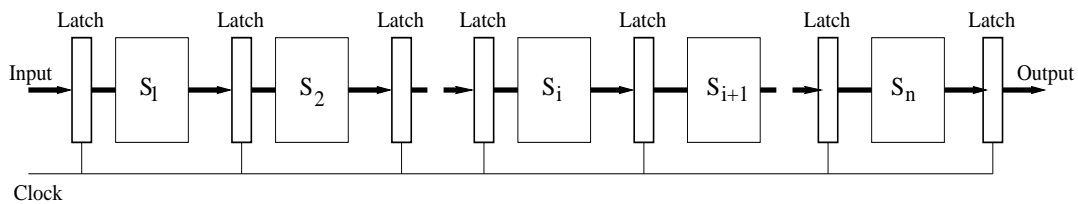


Figure 18: Synchronous pipeline

Utilization of pipeline stages ($S_1, S_2, \dots, S_i, S_{i+1}, \dots, S_n$) in different clock cycles can be specified by reservation table of pipeline function.

In asynchronous pipeline, data transfer from stage S_i to S_{i+1} can only be realized when stage S_{i+1} is ready to accept data from S_i (Figure 19).

This follows a handshaking protocol.

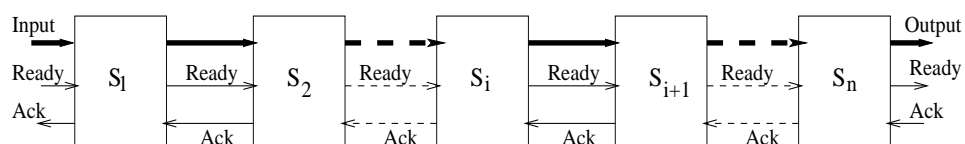


Figure 19: Asynchronous pipeline

0.5 Non-linear Pipeline

A pipeline architecture can reconfigure different functions at different times.

Pipeline architecture of Figure 20 (Figure 21) can realize a function say, F1 that utilizes S_1 , S_2 , and S_3 in 3 consecutive clock/pipeline cycles respectively.

Function F2 may follow S_1 , S_2 , S_3 , S_1 , and S_2 . It requires 5 clock cycles.

Such pipeline structure (allows feedback/feedforward), is a *nonlinear pipeline*.

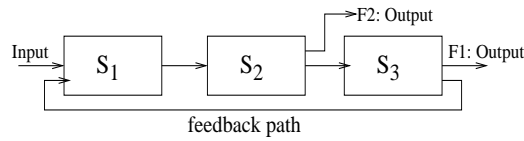


Figure 20: Feedback

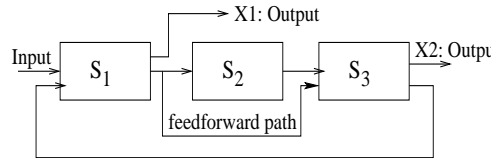


Figure 21: Feedforward

Features of nonlinear pipeline:

- a. These are multifunctional pipelines.
- b. There can be feedback from a later stage j to an earlier stage i of the pipeline (in Figure 20, from stage S_3 to S_1).
- c. There can be feedforward links from pipeline stage i to stage j (in Figure 21, from stage S_1 to S_3).
- d. Pipeline logic controls links/paths that are to be activated at a given cycle.

Lecture 29-30: April 30, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

0.6 Reservation Table

Reservation table describes occupancy of pipeline stages in different clock cycles.

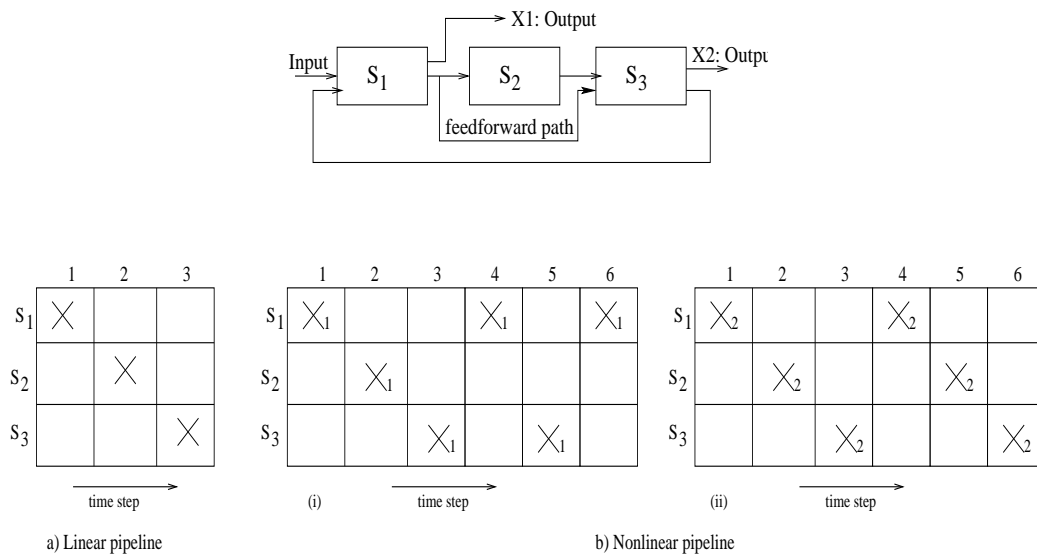


Figure 22: Reservation table a) Linear pipeline b) Nonlinear pipeline

Row: represents resource/pipeline stage. Column: time-slice/pipeline cycle.

Reservation tables of Figure 22 represents a 3-stage (S_1 , S_2 , and S_3) pipeline system.

Linear function X (Figure 22(a)) utilizes each of stages S_1 , S_2 , and S_3 only once.

Multiple entries in a row denote : feedback paths.

A pipeline configuration (hardware structure) can be utilized to realize multiple reservation tables (each corresponds to a pipeline function).

However, a reservation table corresponds to only one pipeline function.

Collision can occur in a nonlinear pipeline to realize a function X .

Collision: When two initiations of X try to use same resource at the same time.

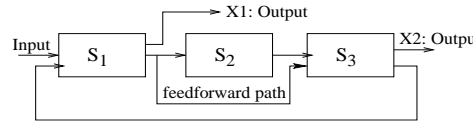


Figure 23 shows collision.

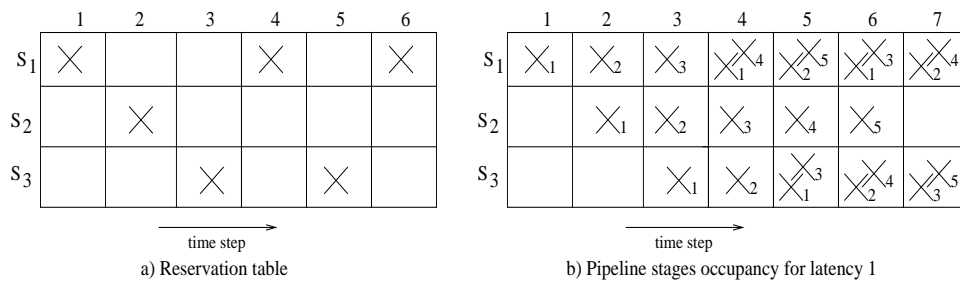


Figure 23: Collision in nonlinear pipeline

In linear pipeline, there is no collision (Figure 24).

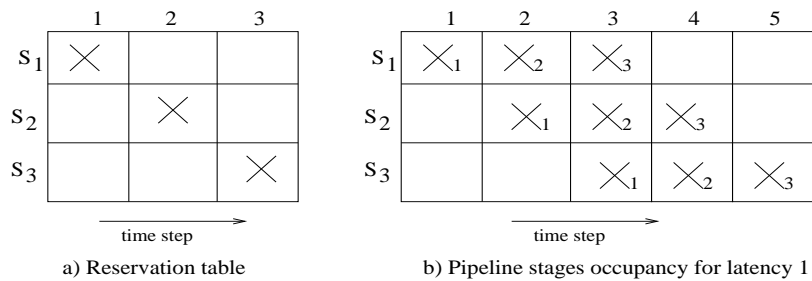


Figure 24: No collision in linear pipeline

0.6.1 Latency analysis

Latency: Number of time units between two initiations of an evaluation.

Forbidden latency: Latency that creates collisions.

Latency sequence: A sequence of permissible non-forbidden latencies between successive task initiations.

Latency cycle: A sequence that repeats itself. Example: latency cycle (1,5) represents infinite latency sequence 1, 5, 1, 5, 1, 5,

Latency cycle represents schedule of initiations avoiding collision.

Average latency of a cycle: Sum of latencies in the cycle divided by number of latencies. Example : average latency for (1,5) is 3. It is 3 for (3) also.

Demands for scheduling of initiations:

1. Collision free scheduling
2. Achieve the shortest average latency

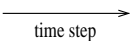
0.6.2 Collision free scheduling

Target is to obtain minimal average latency (MAL). Steps taken to find MAL.

- Step 1. Identify forbidden latencies and set of permissible latencies (can be obtained from reservation table).

In Figure 25, forbidden latencies : 2/3/5. 1 and 4 are permissible latencies.

	1	2	3	4	5	6
s_1	×			×		×
s_2		×				
s_3			×		×	



time step

Figure 25: Nonlinear pipeline

- Step 2. Find collision vector $C = (C_d C_{d-1} C_{d-2} \cdots C_2 C_1)$, $d \leq (n - 1)$

Where n is compute time (number of columns) of reservation table.

$C_i = 1$, if latency i causes a collision (i is forbidden latency),

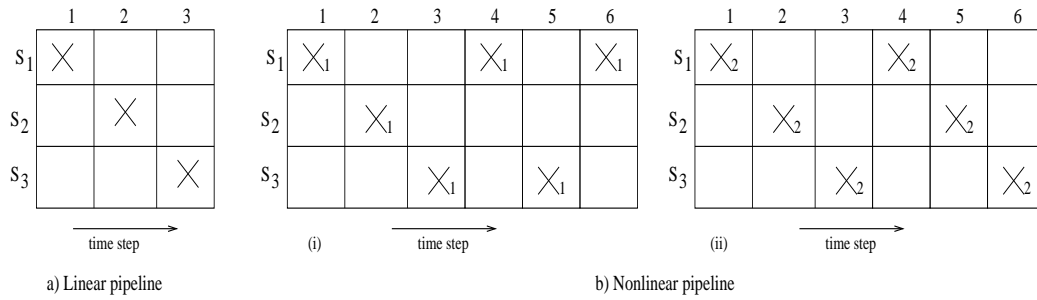
$C_i = 0$, if latency i is permissible.

Most significant (d^{th}) bit of C is always '1'.

For Figure 25, $C = 10110$ ($d = 5 = n-1 = 6 - 1$).

Maximum value of d can be the maximum value of forbidden latency.

C corresponding to reservation table of Figure 22(b)(ii) is 100.



- Step 3. Construct state diagram - permissible transitions - C as initial state.
 - a. For each permissible latency p , compute new state for transition with p ,
 - (i) perform p -bit right shift of current state to get CS and then,
 - (ii) perform OR of CS and C.
 - c. Process (b) is continued for all possible states.

CS defines latencies forbidden for I_{i+1} due to presence of I_{i-1}, I_{i-2}, \dots

For Figure 25, state diagram is Figure 26.

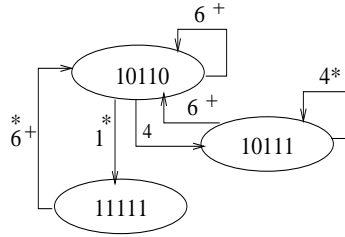


Figure 26: State diagram

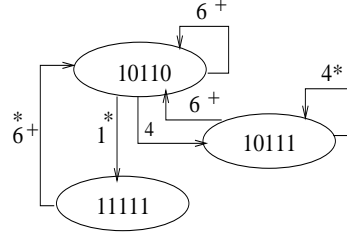
After $p = 1$ (permissible latency) the next state is

$$01011 + 10110 = 11111.$$

All $d+1$ latencies are permissible latencies.

When number of shifts $\geq d+1$, all transitions are redirected to initial state.

‘*’ denotes minimum latency edge. In Figure 26, there are three such latencies.



- Step 4. Determine optimal latency cycles - minimal average latency (MAL).
 - Find legitimate cycles. For Figure 26, the legitimate cycles are (1,6), (6), (4), (4,6), (6,1,6), (4,6,1,6) \dots .
 - Find simple cycles (a latency cycle in which a state appears only once). For this example, simple cycles are (1,6), (4), (6), (4,6).
 - Find greedy cycles (whose edges are made with minimum latencies) from simple cycles. For this example, greedy cycles are (1,6) and (4).
 - Compute average latencies of greedy cycles. Here, these are 3.5 and 4.
 - Find greedy cycles with MAL. For this example, MAL is 3.5.

However, greedy cycle is not sufficient to guarantee the optimality of MAL.

Lower bound of MAL - maximum number of checkmarks in any row of RT.

For this example, lower bound of MAL is 3 (Figure 25).

	1	2	3	4	5	6
s_1	×			×		×
s_2		×				
s_3			×		×	

time step

0.6.3 Scheduling optimization

Insert noncompute delay D in RT (keeping pipeline function intact) to yield an optimal greedy cycle for MAL (Figure 27(a)). Choice of delay: no specific algorithm.

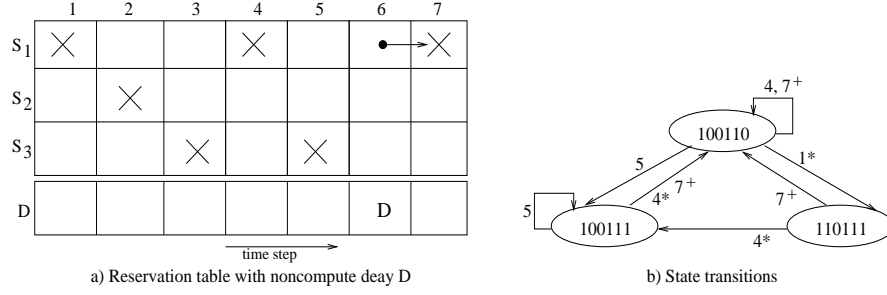


Figure 27: Insertion of noncompute delay

Now, forbidden latencies are 2, 3 and 6. Collision vector is $C = 100110$.

New state transition diagram is shown in Figure 27(b).

There is only one greedy cycle (1, 4, 4). $MAL = \frac{1+4+4}{3} = 3$.

Configuration of new pipeline is in in Figure 28(b). Original one is in Figure 28(a).

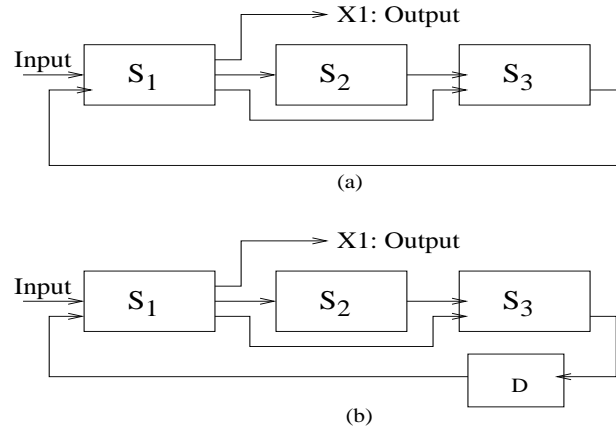


Figure 28: Pipeline architecture