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

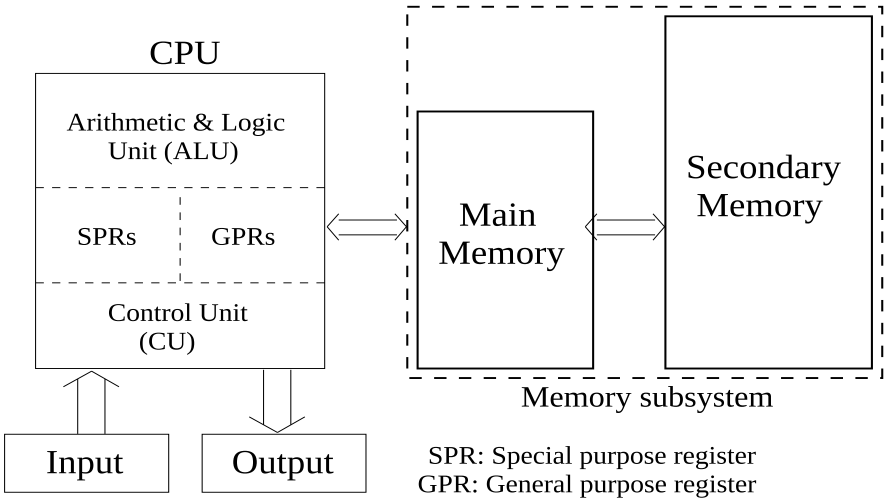
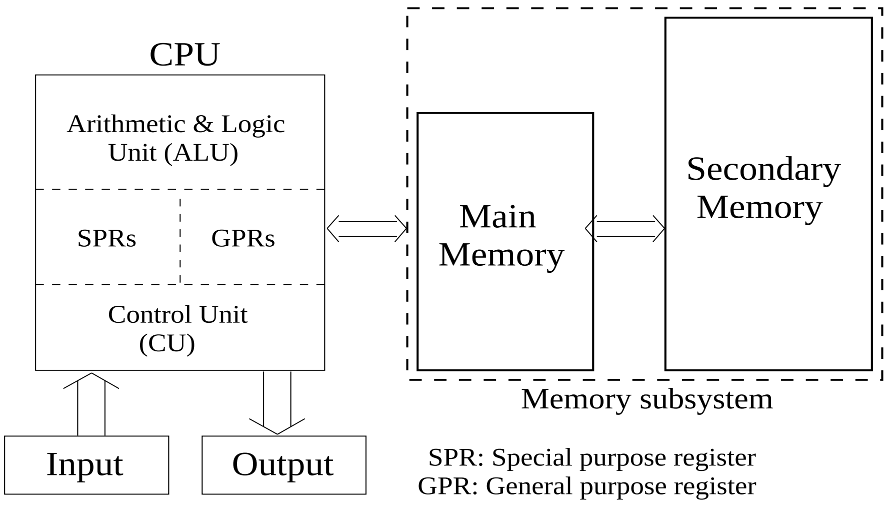

CPU
Arithmetic & Logic Unit (ALU)
SPRs
GPRs
Control Unit (CU)
Input
Output
Main Memory
Secondary Memory
Memory subsystem
SPR: Special purpose register
GPR: General purpose register

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.


CPU
Arithmetic & Logic Unit (ALU)
SPRs
GPRs
Control Unit (CU)
Main Memory
Secondary Memory
Memory subsystem
Input
Output
SPR: Special purpose register
GPR: General purpose register

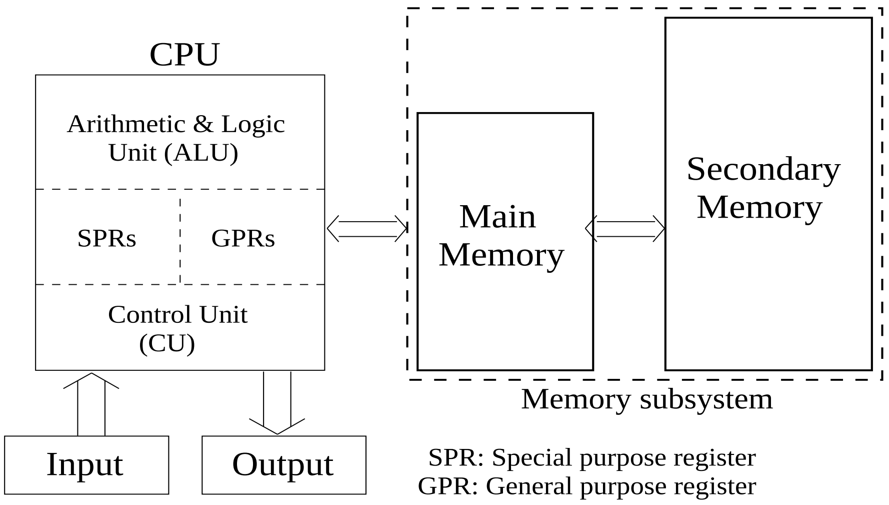
A The central processing unit (CPU)

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.


CPU
Arithmetic & Logic Unit (ALU)
SPRs
GPRs
Control Unit (CU)
Main Memory
Secondary Memory
Memory subsystem
Input
Output
SPR: Special purpose register
GPR: General purpose register

B The main memory (MM)

1. Stores the program (set of instructions) which is to be executed by the CPU.

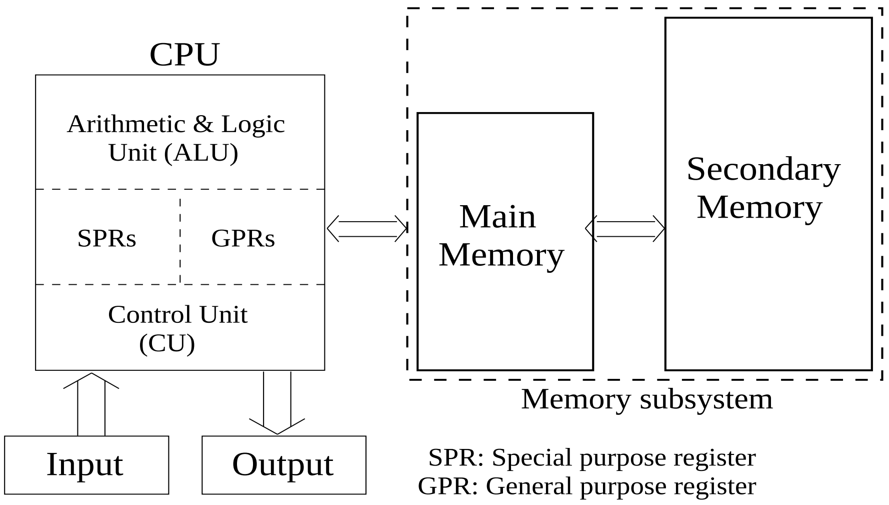
2. An instruction of a program can not be executed by the CPU unless it is brought into MM which is directly interfaced with the CPU.

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 giga[2] 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).

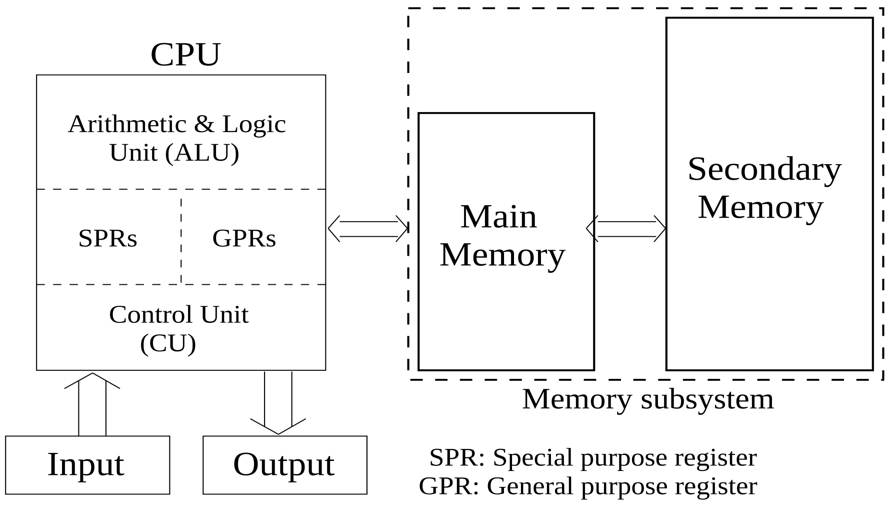
[2] 1 giga bytes (GB) = 10^9 bytes.


CPU
Arithmetic & Logic Unit (ALU)
SPRs
GPRs
Control Unit (CU)
Main Memory
Secondary Memory
Memory subsystem
Input
Output
SPR: Special purpose register
GPR: General purpose register

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)).


CPU
Arithmetic & Logic Unit (ALU)
SPRs
GPRs
Control Unit (CU)
Main Memory
Secondary Memory
Memory subsystem
Input
Output
SPR: Special purpose register
GPR: General purpose register

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/ASCII[3], etc.

The speed variation of input/output devices ranges from few bytes to mega bits[4].

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).

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

[4]Keyboard 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).

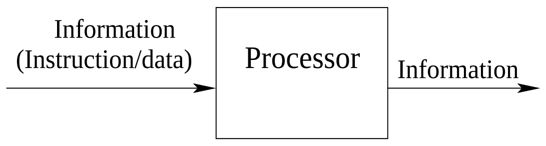


Figure 2: The computer

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$\times$50 sq ft. It used about 18,000 vacuum tubes (Figure 3).

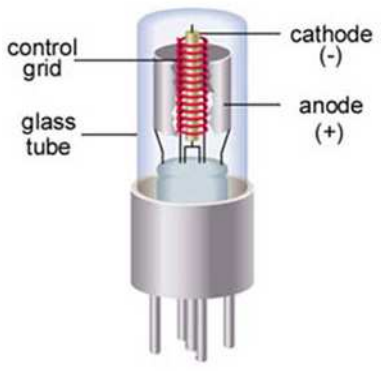


Figure 3: Vacuum tube

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.
- Programing 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).

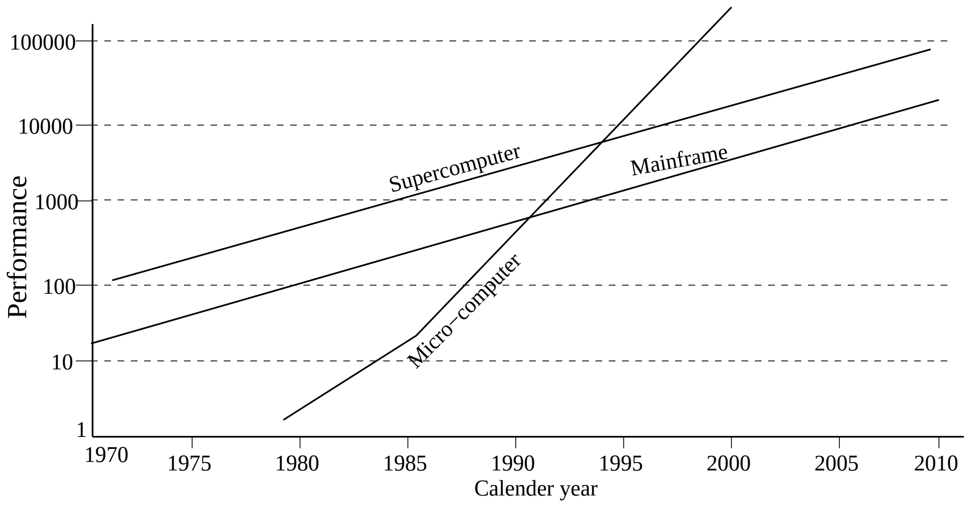


Figure 4: Performance trend

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).

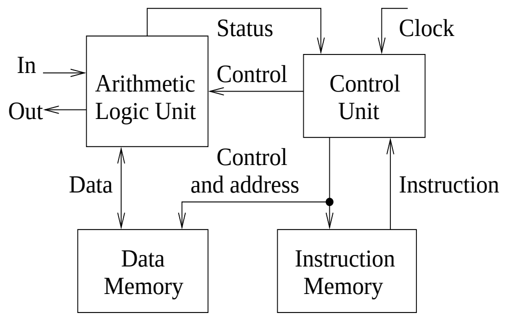


Figure 1: The harvard architecture

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.

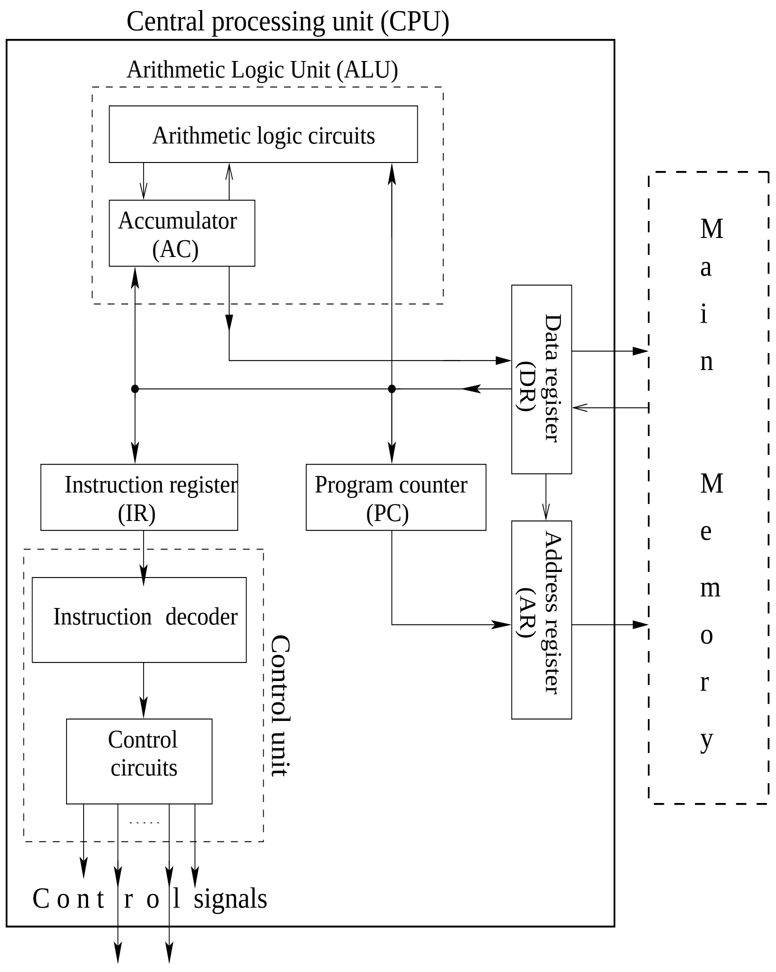


Figure 2: The basic von Neumann architecture

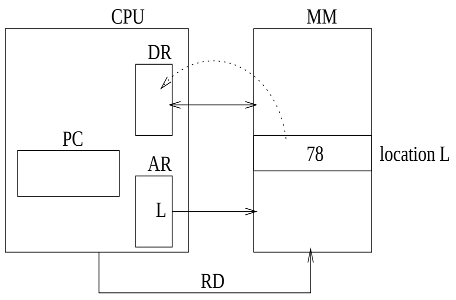


Figure 3: Memory read

- **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 archutecture, there is no explicit distinction between inst and data.

That is,

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

  This increases the complexity of debugging an erroneous program.

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

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

0.1.2 The tags

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

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

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

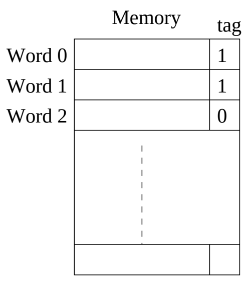


Figure 4: The tag

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

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

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

The tags are considered in RISC architecture.

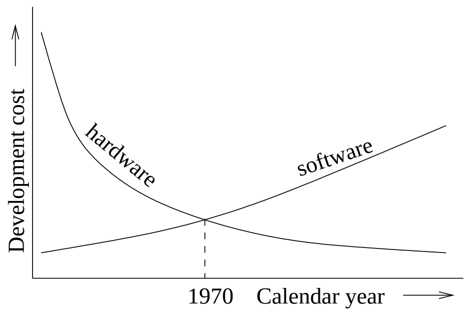


Figure 5: Hardware vs software cost

0.1.3 Instruction sequencing

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

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

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

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

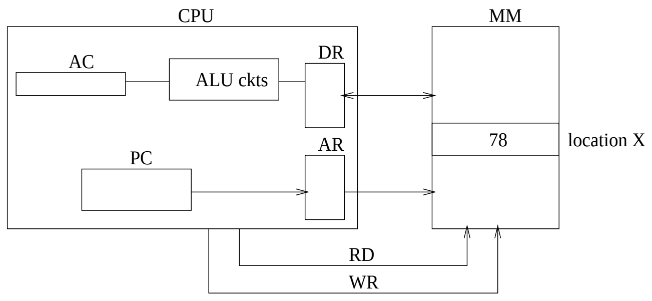


Figure 6: Macro instruction execution

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

Other operands of the instruction are implicit.

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

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

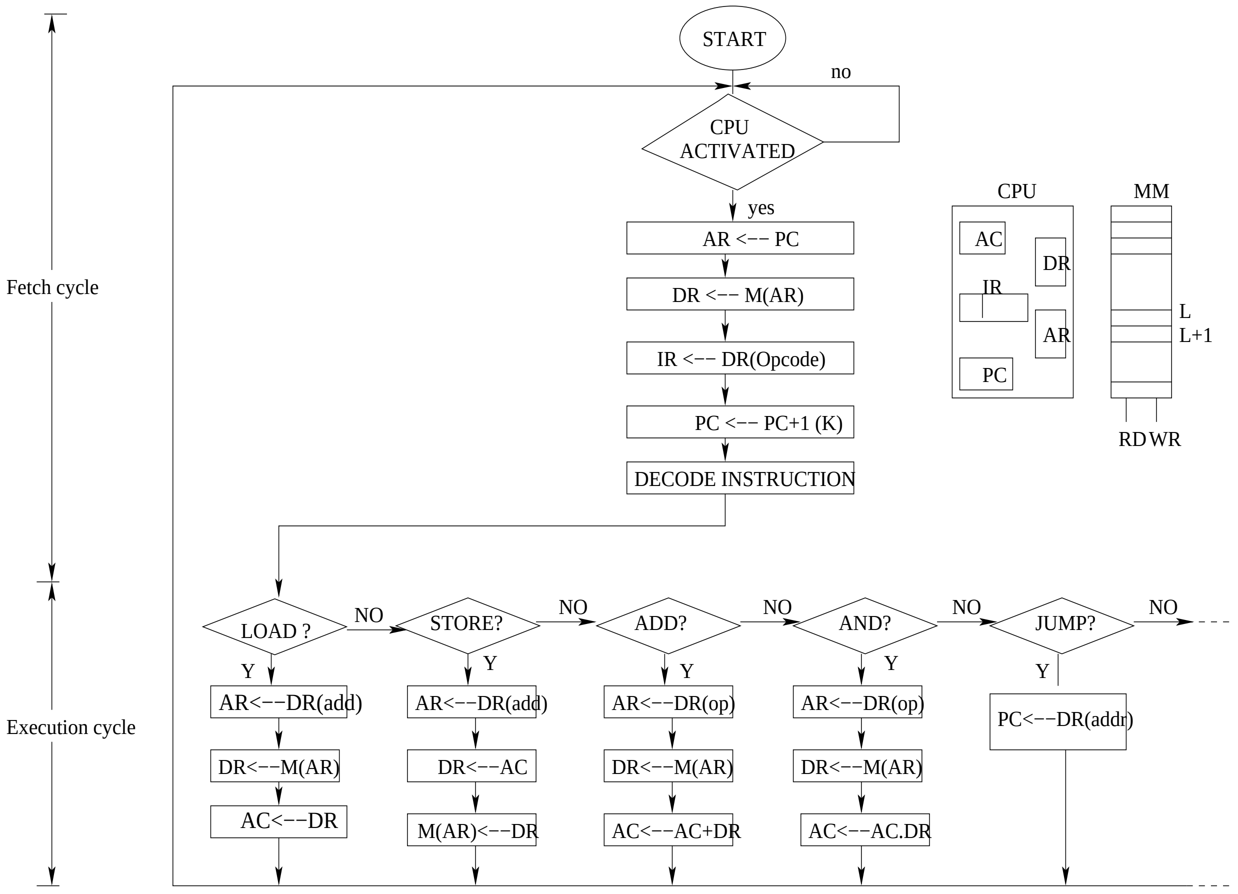
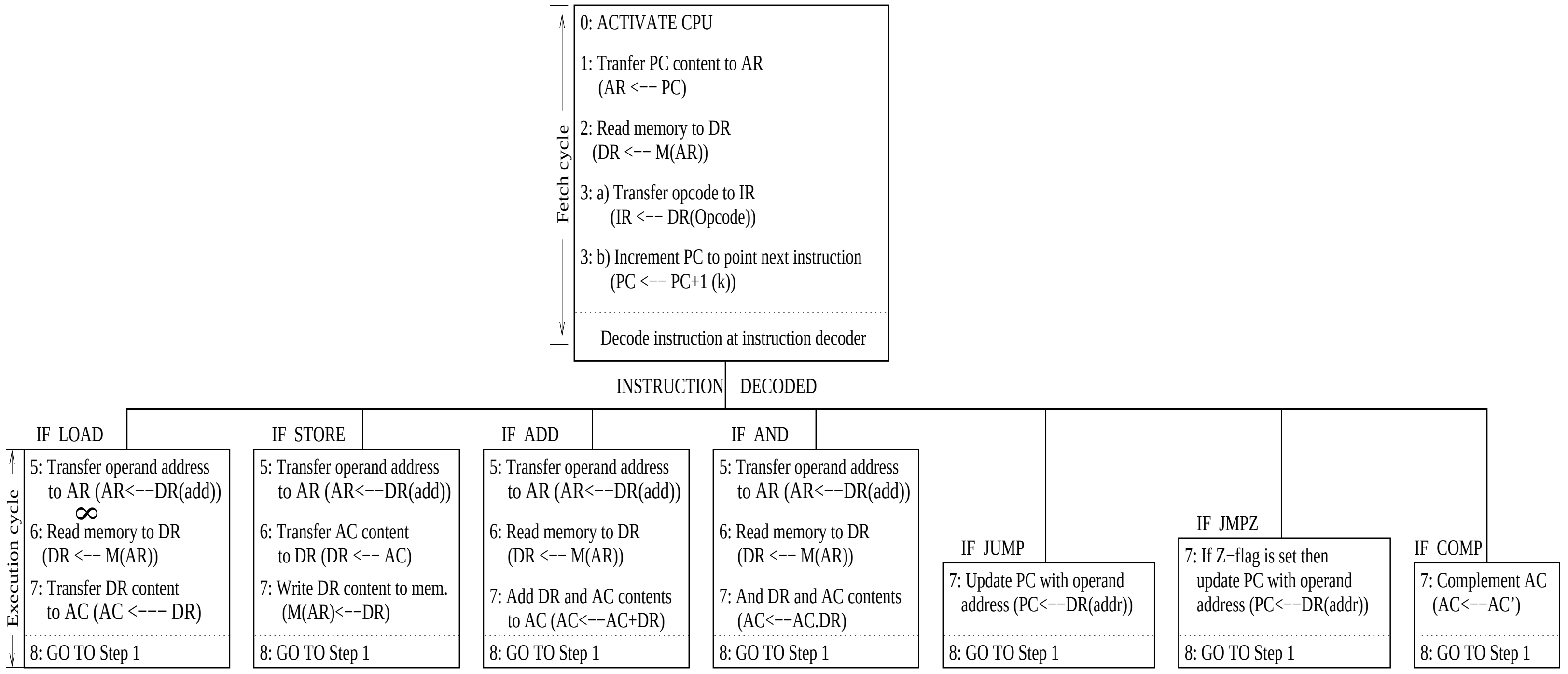


Figure 7: Flow chart



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

Figure 8: Execution of von Neumann macro instructions

Memory

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

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

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

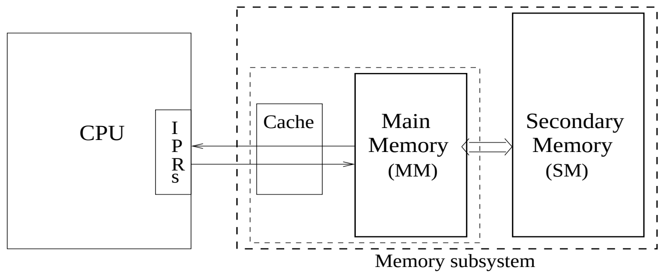
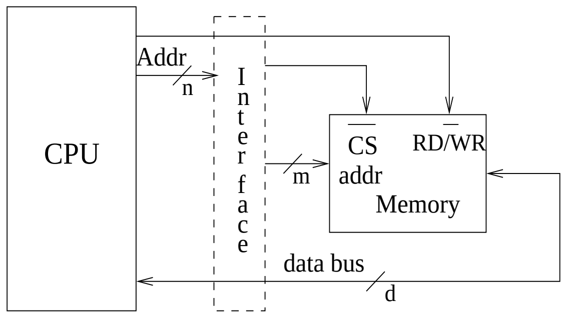
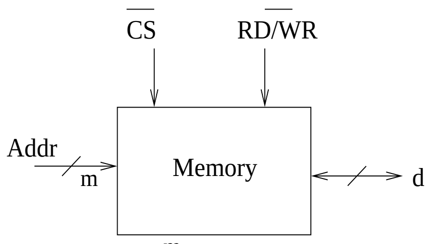


Figure 9: Computer memory subsystem



a) Memory interfacing



b) 2^m x d memory

Figure 10: Memory interfacing

0.2 Memory Interfacing

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

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

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

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

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

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

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

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

0.3 Memory Design

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

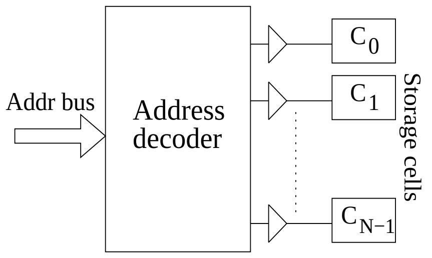


Figure 11: Memory module I

Bit-organized

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

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

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

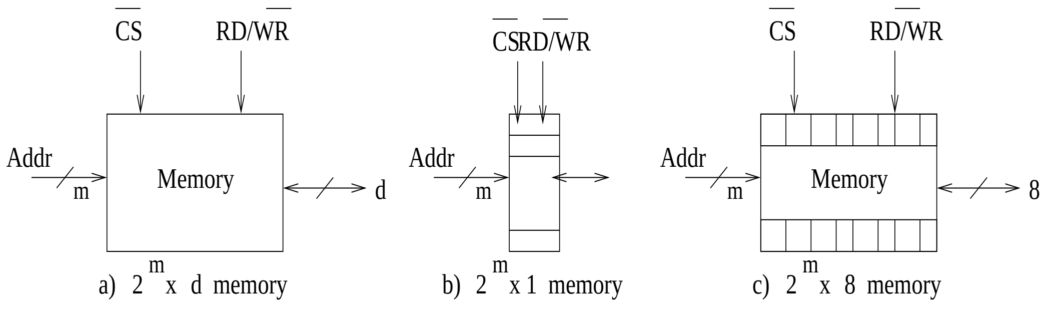


Figure 12: Memory organized

** Design $1\times$ 1 memory

Constructing large memory module

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

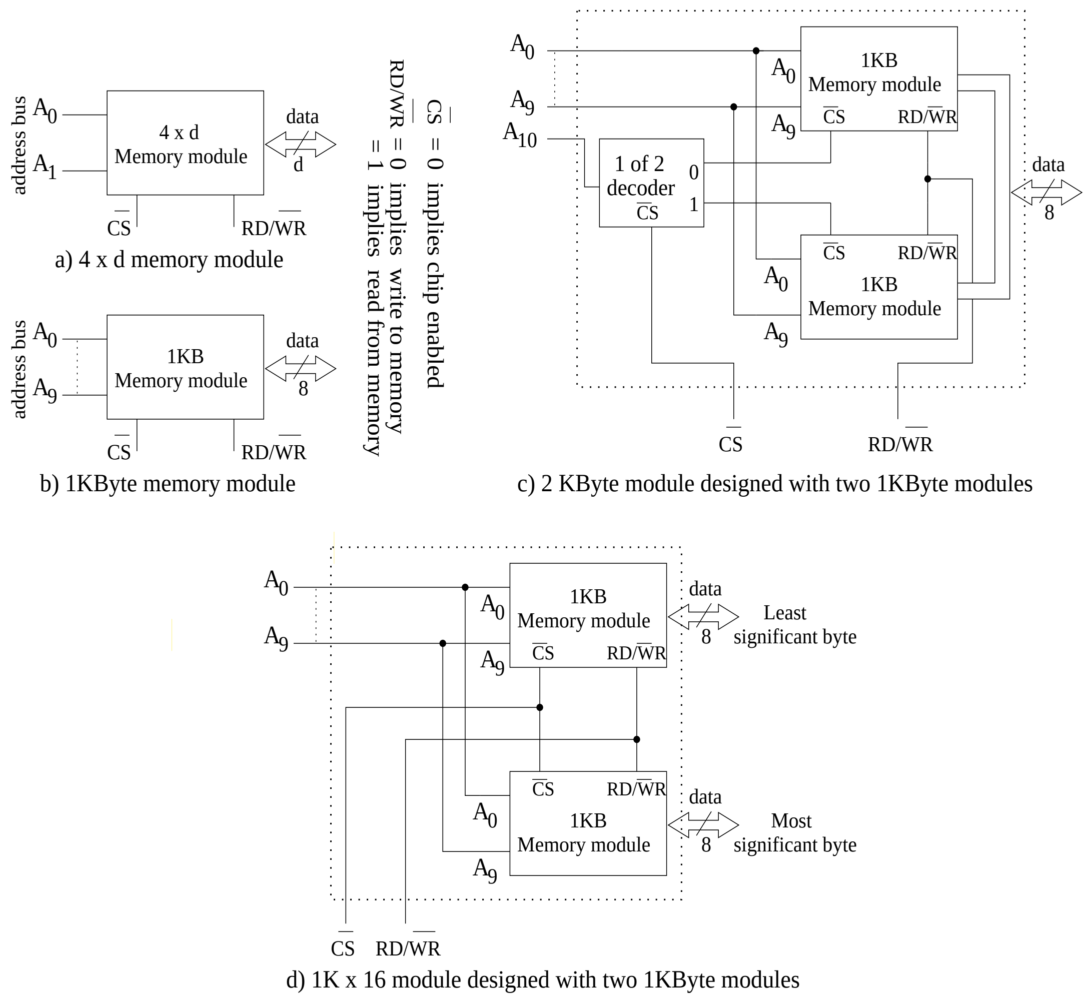

address bus
A0
A1
4 x d
Memory module
data
d
CS
RD/WR
a) 4 x d memory module
A0
A9
1KB
Memory module
data
8
b) 1KByte memory module
CS = 0 implies chip enabled
RD/WR = 0 implies write to memory
= 1 implies read from memory
A10
1 of 2
decoder
0
1
c) 2 KByte module designed with two 1KByte modules
Least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

Figure 13: Memory arrays

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

1× 1 memory

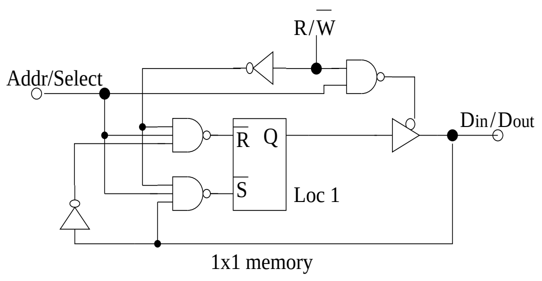


Figure 14: 1x1 memory

Verify the design as per the following table.

Table 1: Verification for 1x1 memory

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

$1\times$ 2 memory

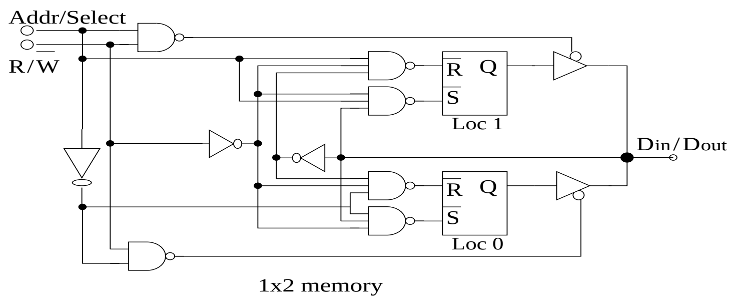


Figure 15: 1x2 memory

Verify the design as per the following table.

Table 2: Verufication for 1x2 memory

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

2× 2 memory

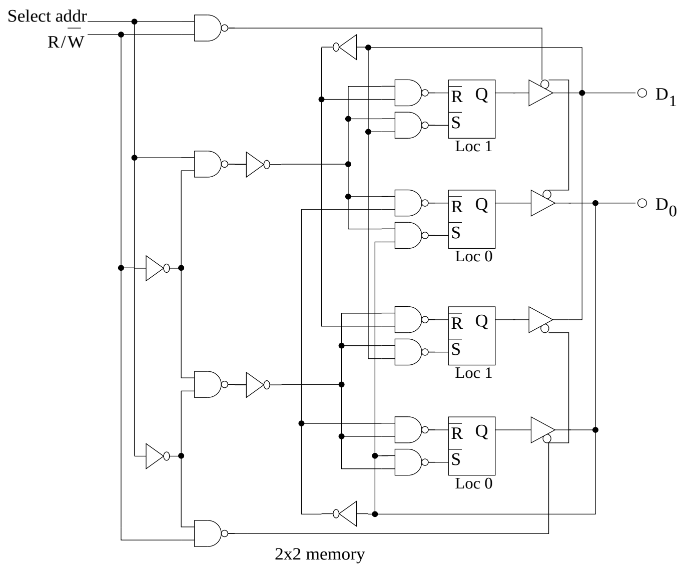


Figure 16: 2x2 memory

Verify the design as per the following table.

Table 3: Verification for 2x2 memory

No	Select	R/$\overline{W}$	$D1_{in}$	$D0_{in}$	$D1_{out}$	$D0_{out}$	Verify
1	1	0	1	0	-	-	Write 10 in Loc-1
2	0	0	0	1	-	-	Write 01 in Loc-0
3	1	1	-	-	1	0	Read 10 from Loc-1
4	0	1	-	-	0	1	Read 01 from Loc-0
5	1	0	0	0	-	-	Write 00 in Loc-1
6	0	0	1	1	-	-	Write 11 in Loc-0
7	1	1	-	-	0	0	Read 00 from Loc-1
8	0	1	-	-	1	1	Read 11 from Loc-0

Lecture 4-5: February 19, 2021
Computer Architecture and Organization-I
Biplab K Sikdar

0.0.1 Extension of basic organization

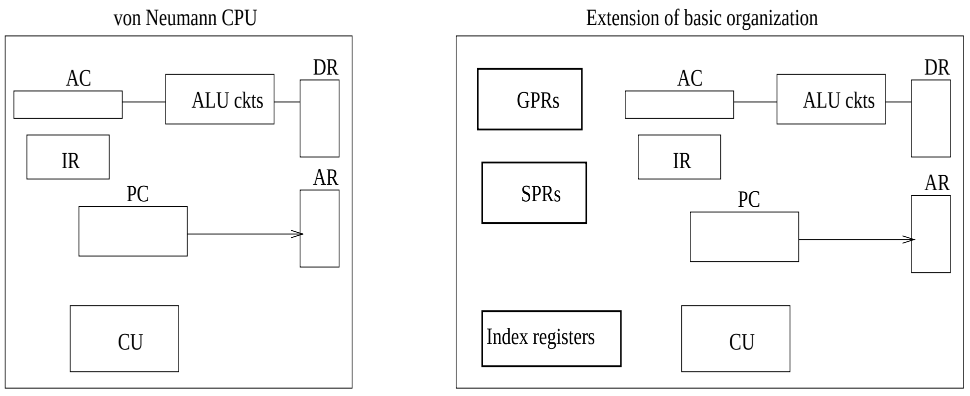


Figure 1: Extension of basic von Neumann architecture

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, $\cdots$. 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, $\cdots$, 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.

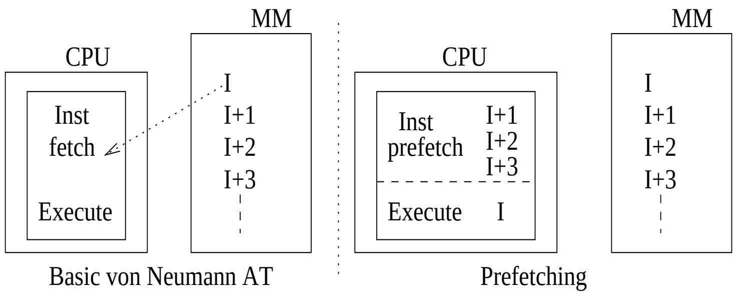


Figure 2: Instruction prefetching

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).

hese 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.

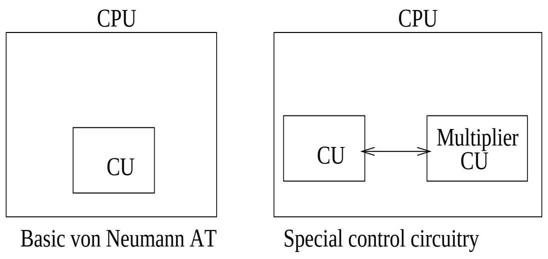


Figure 3: Special control circuitry

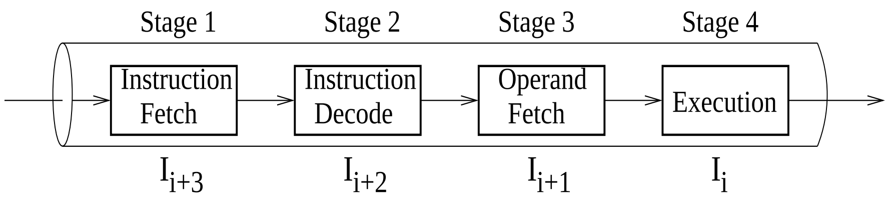


Figure 4: Pipelining

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.

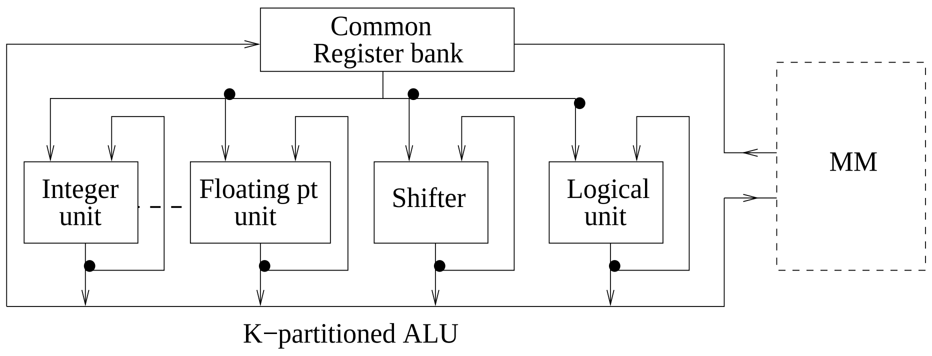


Figure 5: CPU with K-partitioned ALU

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

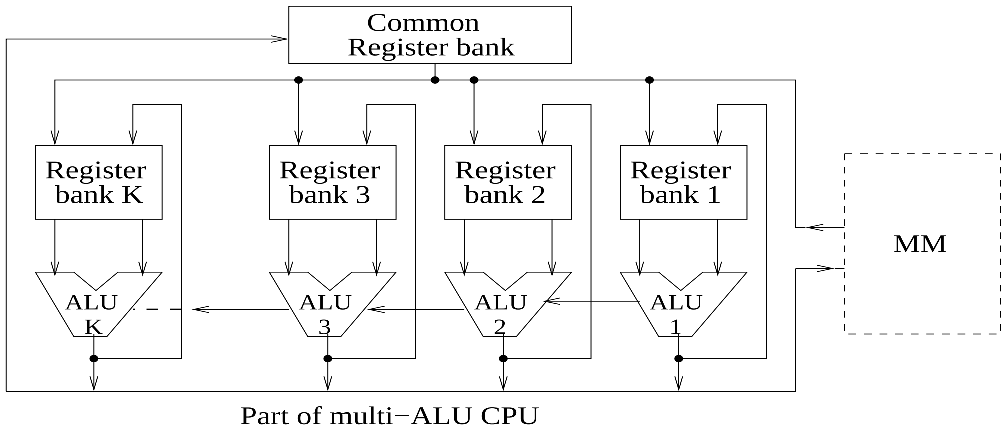


Figure 6: CPU with replicated ALUs

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 intruction set can not be arbitrary.

The insruction set architecture design follows a few guidelines.

0.1 Instruction Set Design

Following requirements are to be fullfilled 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 arithmatic 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 implementaion 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) Arithmatic 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

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

Follow Figure 7.

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

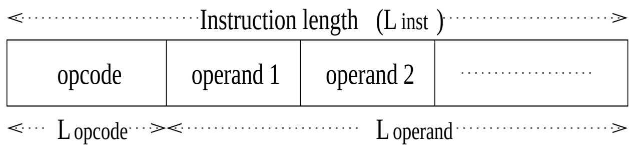


Figure 7: Instruction format

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 delevering 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

a) Optimization of the length of operand field, and

b) 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 legth 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

$$\text{ADD R}_i\text{, R}_j \text{ (R}_i\leftarrow\text{ R}_i\text{ + R}_j\text{)}$$

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, ADD X (AC← 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 (accunulator).

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, $\cdots$.*

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

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

is an example of 3-addressed instruction.

0.3.1.2 2-address instruction

Example, in the follwing 2-addressed instruction

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

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

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

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:

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

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 interms 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.

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

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

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

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

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

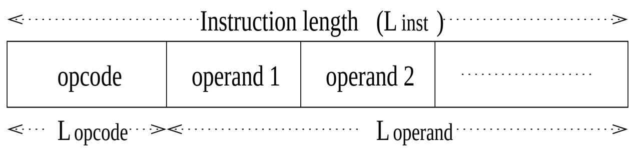
The solution in 0-addressed m/c

```
PUSH  w
PUSH  x
SUB
PUSH  y
ADD
POP  Z
```

requires 6 instructions.

Lecture 6-- 1: February 23, 2021
Computer Architecture and Organization-I
Biplab K Sikdar

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 with in the instruction operand fields as constant.

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

opcode
operand data
LDI
53

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.

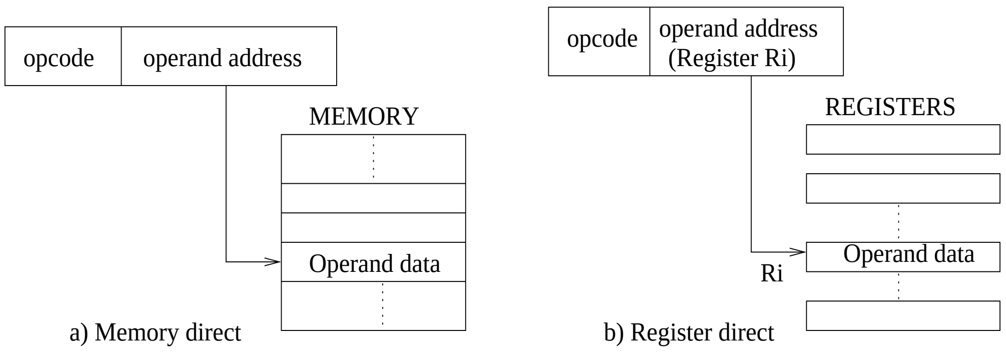


Figure 3: Direct addressing mode

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.

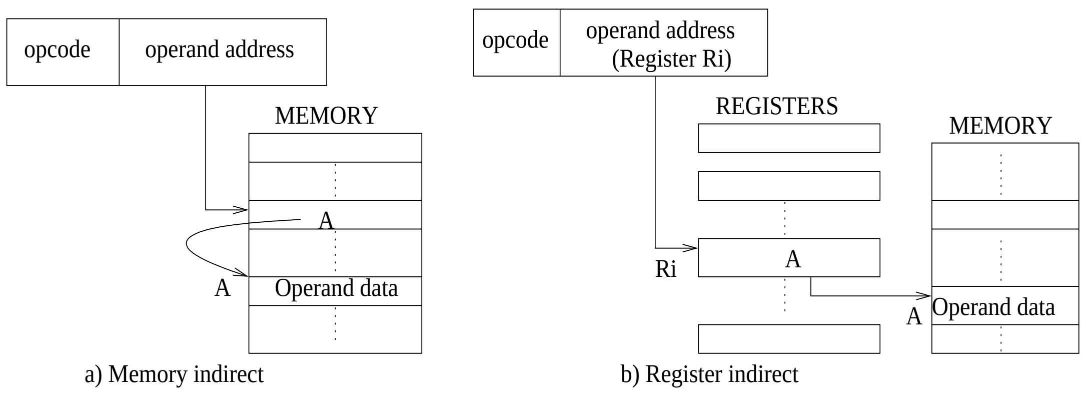


Figure 4: Indirect addressing mode

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

Example: memory indirect -

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

For register direct or inderect, 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.

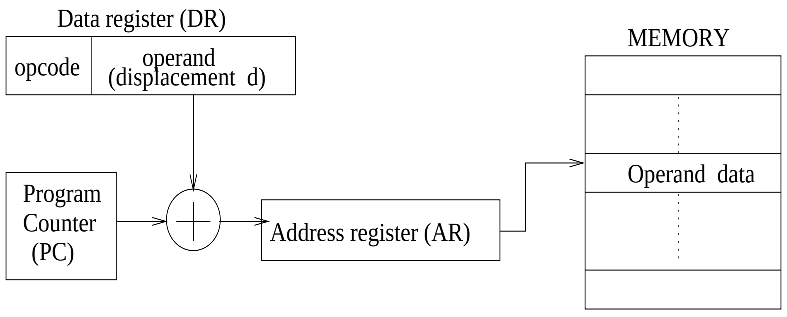


Figure 5: Relative addressing mode

Example:

Store Rel d

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

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

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.

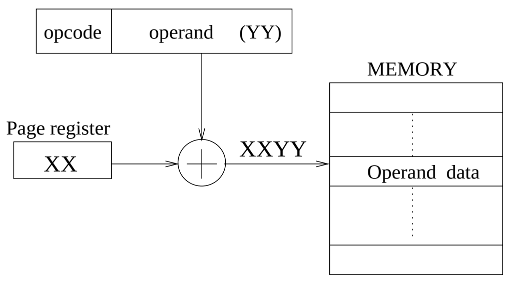


Figure 6: Page addressing

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 explicitlt 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.

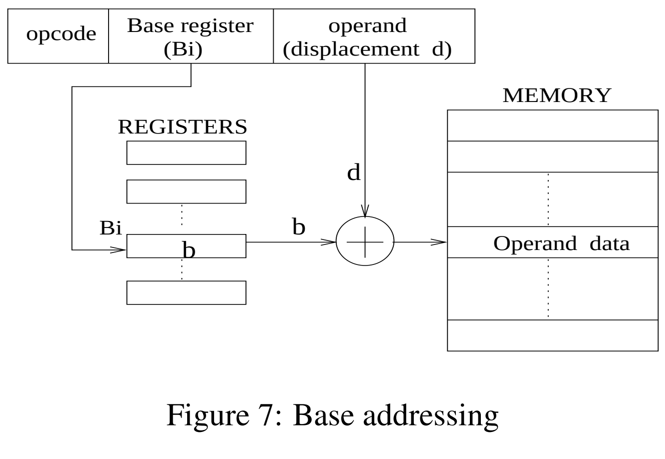


Figure 7: Base addressing

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.

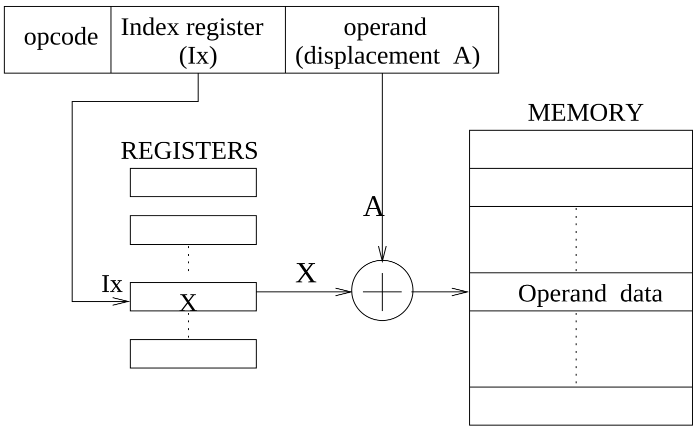


Figure 8: Indexed addressing

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

It is effective when we need to perform iterative operations.

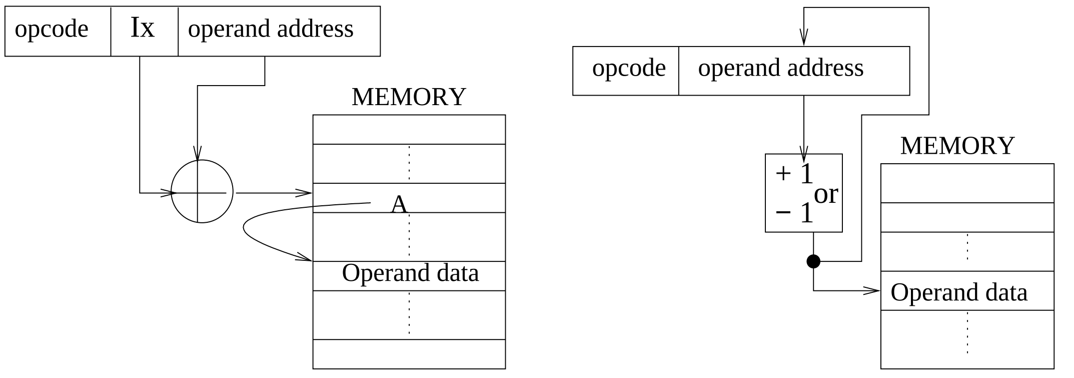
Example: In a program if we need to execute the same operation I iteratively on different operands at A, A+1, A+2, $\cdots$, then with the help of indexing this can be realized in a simpler way. In indexing,

a) The I's operand field is set to A,

b) The index register (IX) chosen is initialized to 0,

c) After each I operation, the IX is incremented by 1.

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



a) Indexing with memory indirect   b) Auto−increment/decrement memory direct

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.

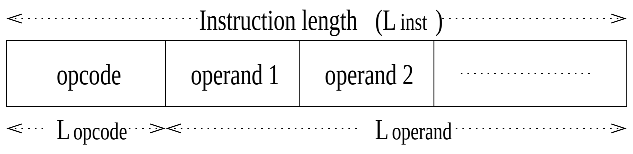
The x86 allows a variety of addressing modes such as

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.

Lecture 7-8: February 26, 2021
Computer Architecture and Organization-I
Biplab K Sikdar

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).

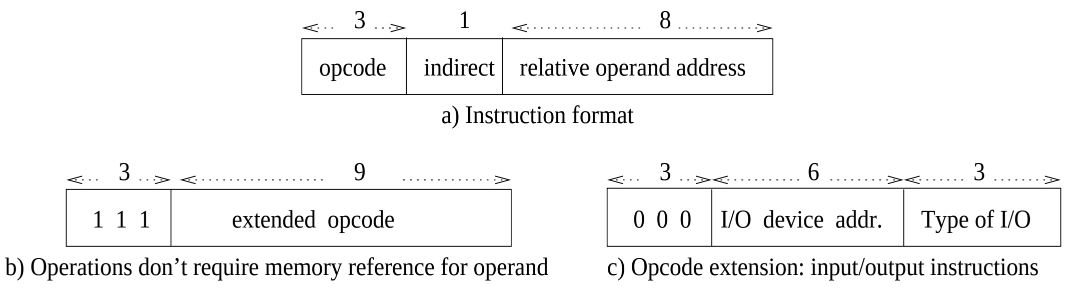
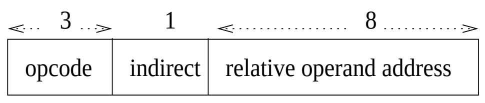
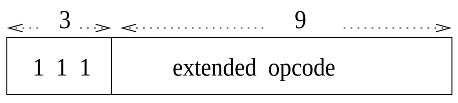


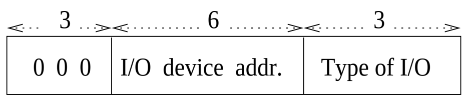
Figure 10: Instruction format of PDP-8



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:

$$\text{ADD } r_1, r_2$$

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.

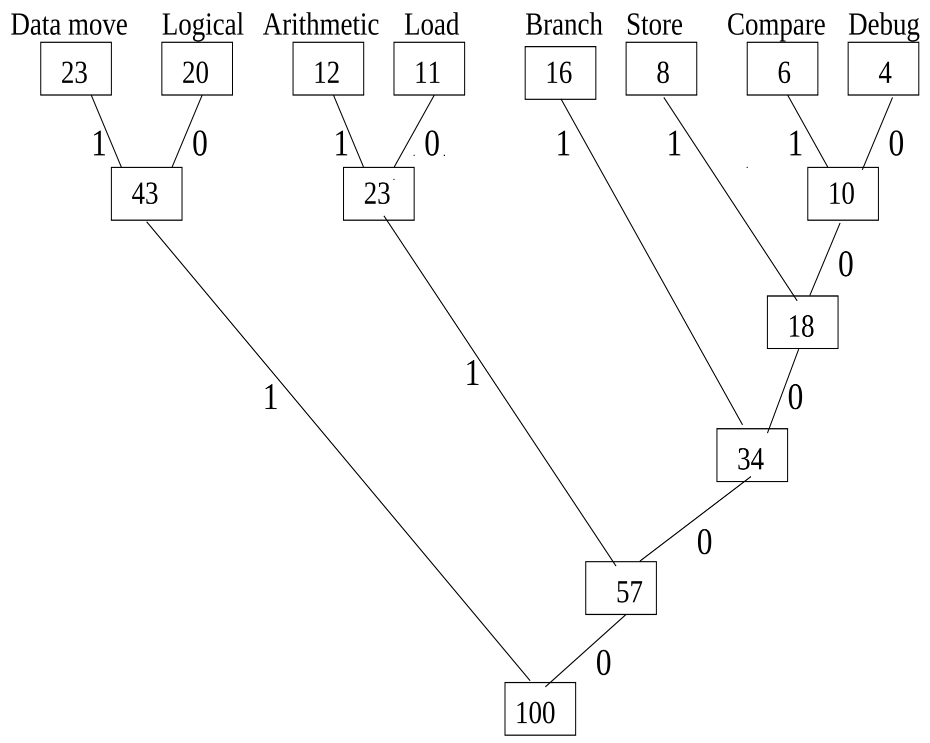


Figure 11: Frequency dependent opcode optimization

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

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

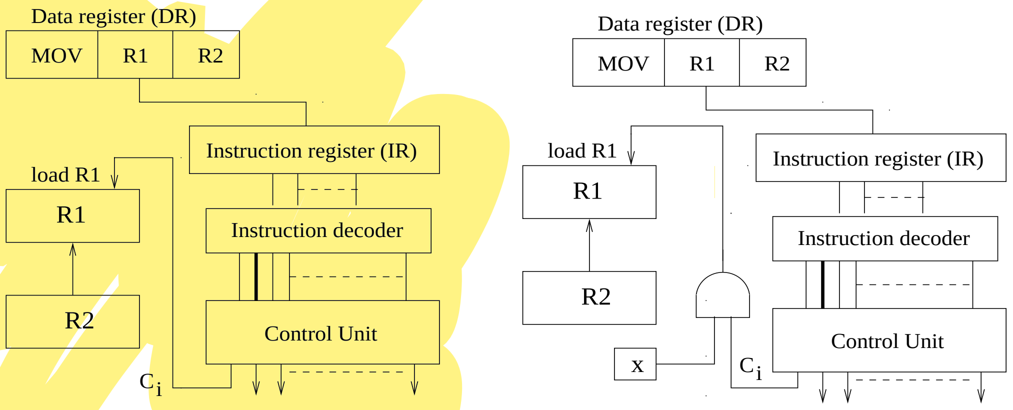
Execution of an (macro) instruction involves execution of a sequence of micro-operations.

0.7.1 Data movement

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

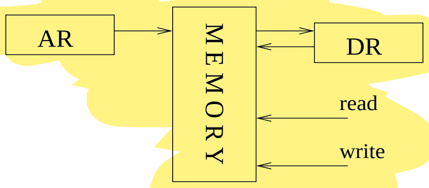
Implementation of register to register data transfer instruction is shown in Figure 12(a).

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



(i) MOV R1, R2 (ii) If x then MOV R1, R2

a) Register to register data transfer



read : DR $\leftarrow$ M(AR), memory to register transfer
write : M(AR) $\leftarrow$ DR, register to memory transfer

b) Data transfer between register and memory

Figure 12: Data transfer instruction implementation

However, the conditional data movement

$$\text{if } x \text{ MOV R1, R2} \qquad (\text{R1} \leftarrow \text{R2})$$

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).

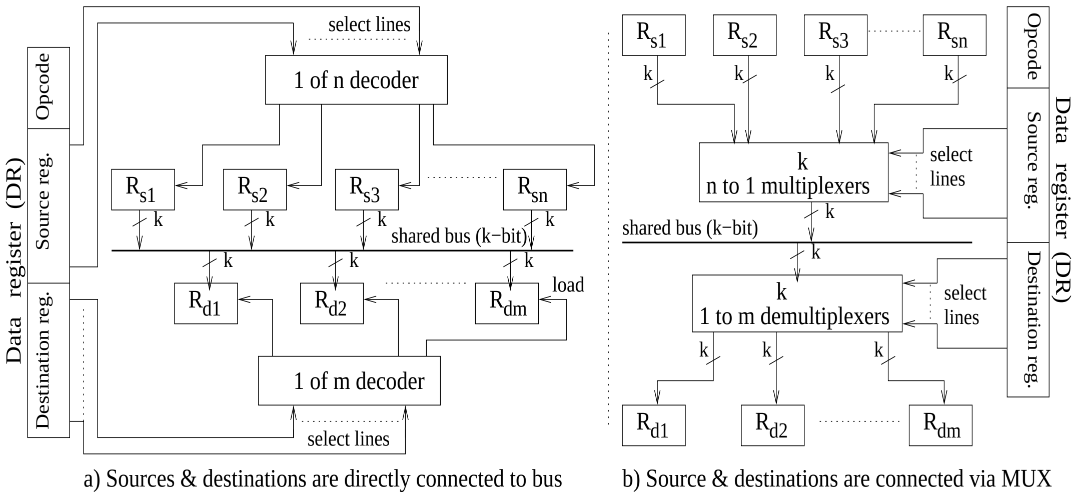


Figure 13: Data transfer from many sources to many destinations

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.

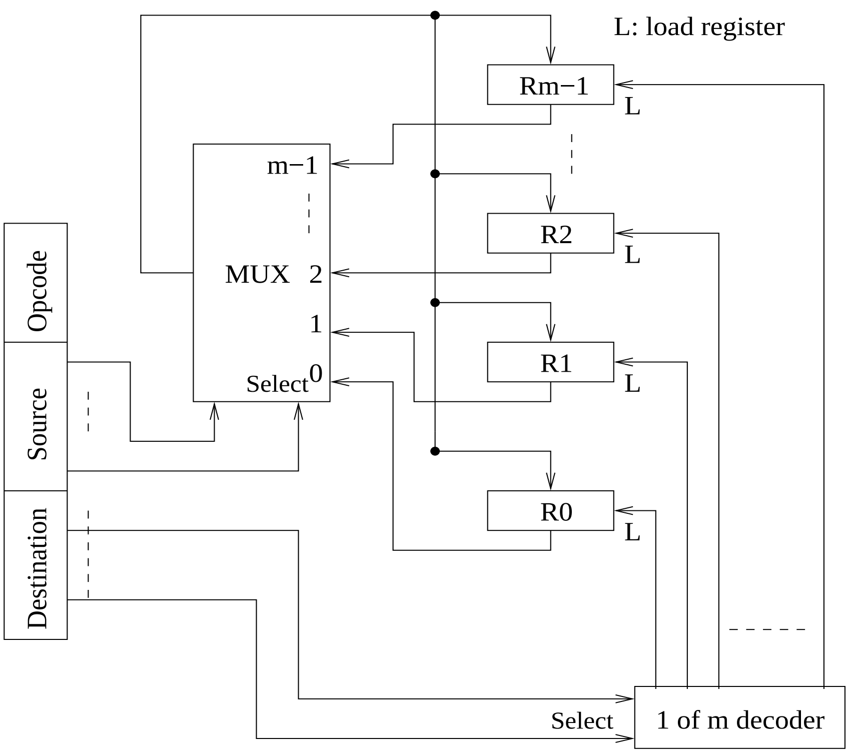


Figure 14: Data transfer among same set of registers

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 (operand).

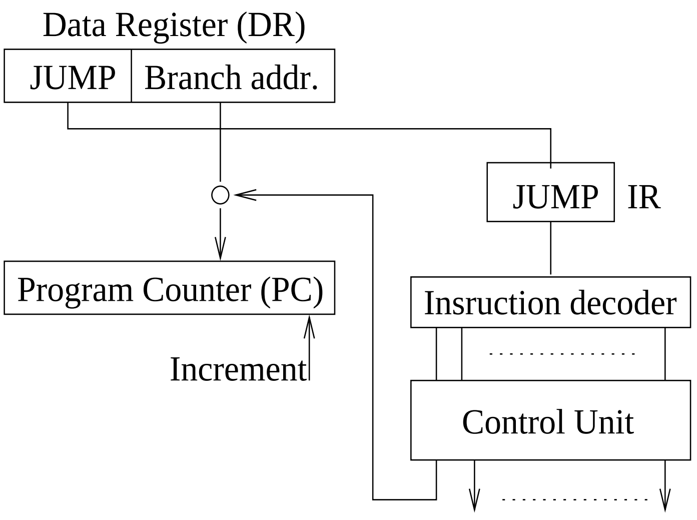


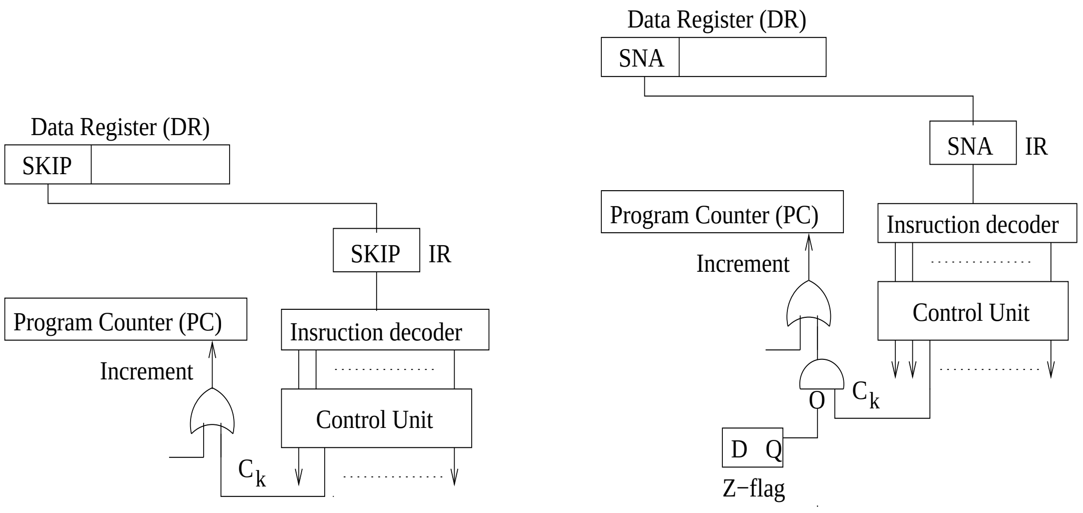
Figure 15: Branch control

Skip implies one instruction is to be skipped.

$$\begin{array}{r|l}
 & \vdots \\
\text{L:} & \text{SKIP} \\
\text{L+1:} & \text{ADD}\ \ \text{X, Y, Z} \\
\text{L+2:} & \text{I}_i \\
 & \vdots
\end{array}$$

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

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



a) Unconditional skip instruction implementation b) Conditional skip instruction implementation

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 satified, then register to register to data transfer

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

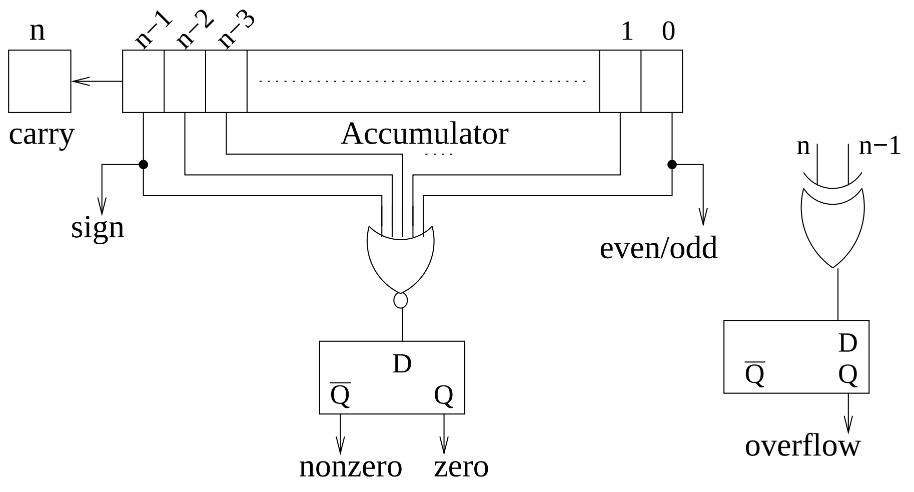


Figure 17: Setting condition flags

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

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

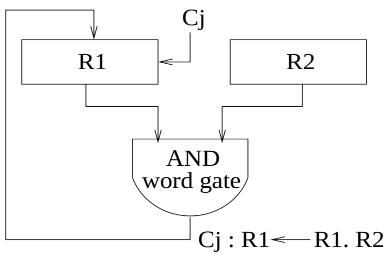


Figure 18: Logical micro-operation implementation

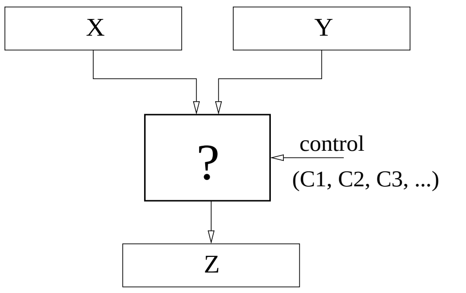
Implementation of logical microoperation is shown in Figure 18.

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

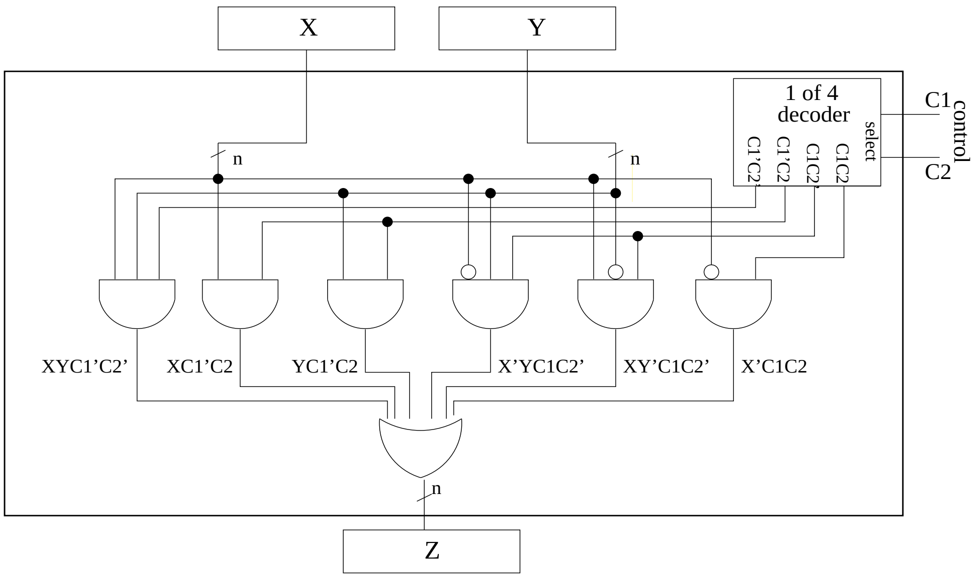
The not operates as Z $\leftarrow$ X$'$.

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

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



a) Implementation of logical instruction Z ← X op Y



b) Implementation of four logical functions with word gate

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.

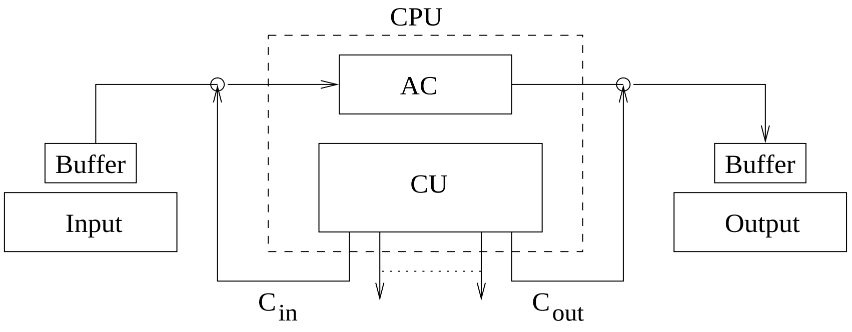


Figure 22: Input-Output instruction implementation

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).

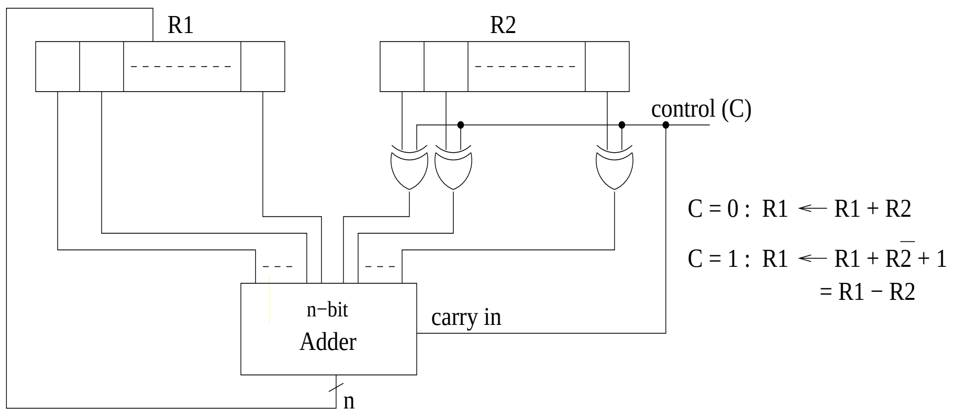


Figure 23: Adder/subtractor

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

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

Design adders

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

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

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

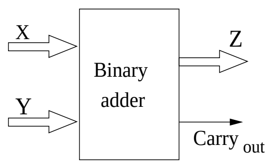


Figure 24: Binary adder

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.

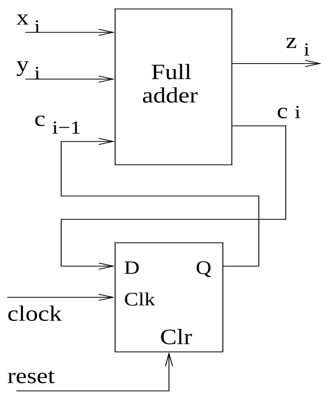


Figure 25: Serial adder

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 propagares 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.

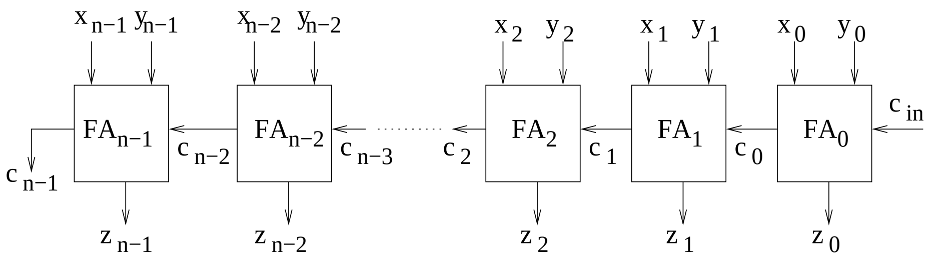


Figure 26: Parallel adder

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 propagare (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,

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

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

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

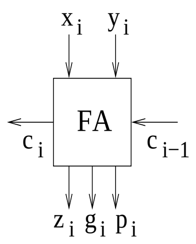
$$c_0 = g_0 + p_0 c_{in}$$

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

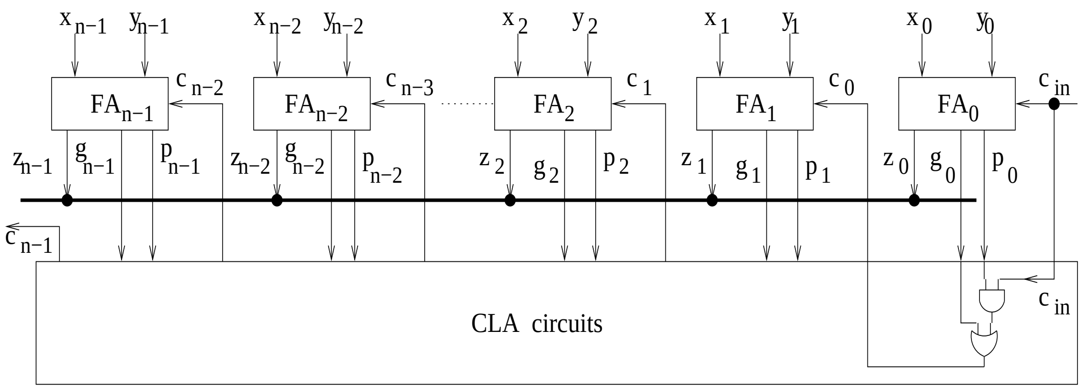
$$c_2 = g_2 + p_2 c_1 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_{in}$$

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

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



a) CLA full adder block



b) n−bit carry−look−ahead adder

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_is and g_is are generated. During second d carrys (c_is) are calculated. In third d, zs (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

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

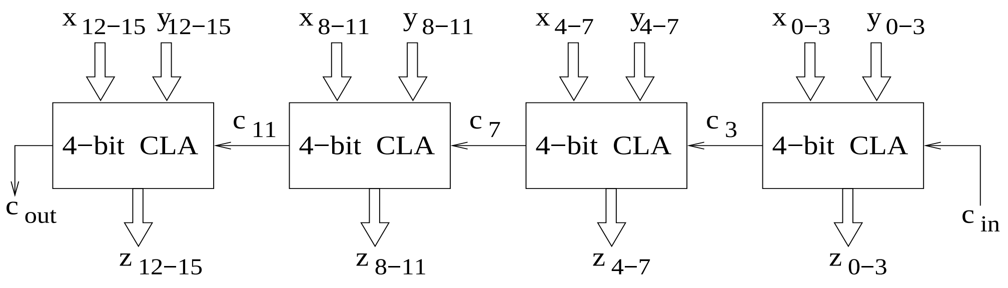


Figure 28: Cascaded carry look ahead adder

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).

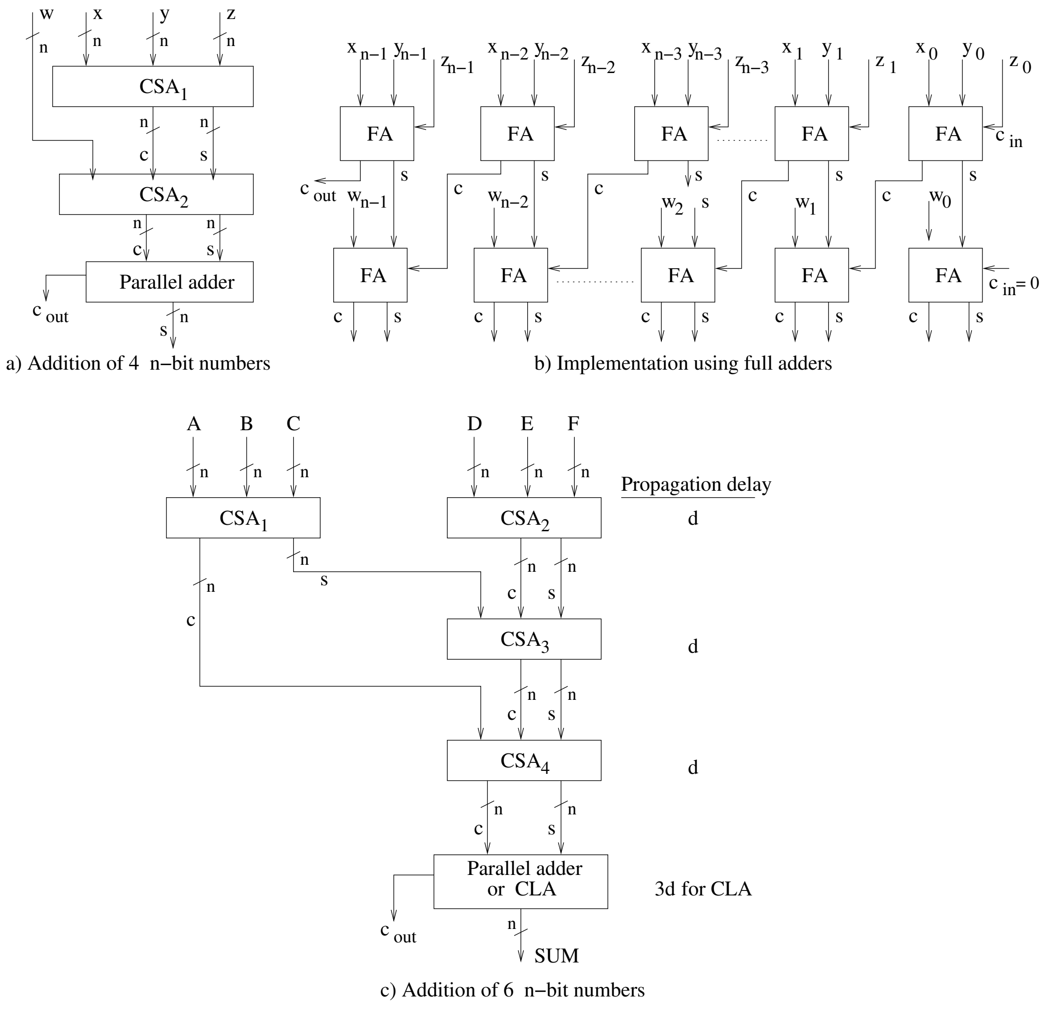


Figure 29: Carry save addition

0.9 Multiplication Instruction Implementation

0.9.1 Multiplication in sign magnitude

Simplest implementation of fixed point multiplication instruction is using conters.

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 PUODUCT = 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
000000**0**	0000000	P_1
00000**11**	0010100	P_2
0000**110**	0101000	P_3
000**1100**	1010000	P_4
00**00000**	0000000	P_5
0**110010**	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, $\cdots$, 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
000000**0**	0000000	P_1
00000**11**	0010100	P_2
0000**110**	0101000	P_3
000**1100**	1010000	P_4
00**00000**	0000000	P_5
0**110010**	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] $\leftarrow$ multiplicand[n-2:0]; Q[n-2:0] $\leftarrow$ 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	$110\underline{1}$	1010	0
$+$		1010			
	0	1010			
RS	0	0101	$011\underline{0}$		1
RS	0	0010	$101\underline{1}$		2
$+$		1010			
	0	1100			
RS	0	0110	$010\underline{1}$		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: $\underline{y_{n-1} = x_{n-1} = 0}$: Algorithm 0.1 can produce the correct result.
- Case II: $\underline{y_{n-1} = 1 \text{ and } x_{n-1} = 0}$: (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: $\underline{y_{n-1} = 0 \text{ and } x_{n-1} = 1}$: 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: $\underline{y_{n-1} = x_{n-1} = 1}$: 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] $\leftarrow$ multiplicand[$n$-1:0]; Q[$n$-1:0] $\leftarrow$ 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	$110\underline{1}$	1010	0
$+$		1010			
	0	1010			
RS	0	1101	$011\underline{0}$		1
RS	0	1110	$101\underline{1}$		2
$+$		1010			
	1	1000			
RS	1	1100	$010\underline{1}$		
$+$		1010			
	1	0110			
RS	1	1011	$001\underline{0}$		
$-$		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×Q, where Q is the multiplier and

$$Q = \overset{i=4}{0}\ \ \overset{3\,2\,1\,0}{1110} = 14.$$

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

Similarly, for the multiplier

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

P = +2^{10}×M - 2^7×M + 2^5×M - 2^1×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_ix_{i-1}.

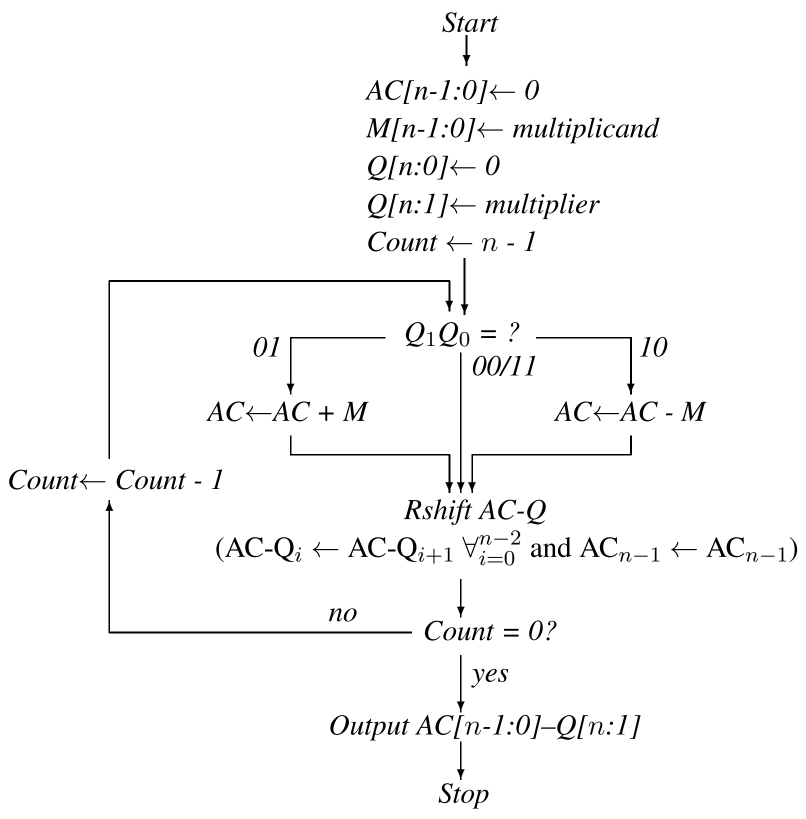
- If x_ix_{i-1} $\begin{array}{ll} = 01, & multiplicand\,Y\ is\ added\ to\ partial\ product\ PP_{i-1}. \\ = 10, & Y\ is\ subtracted\ from\ PP_{i-1}. \end{array}$

  Folled by a left shift to get PP$_i$. Here, PP$_i$ defines product of Y and $x_ix_{i-1}\cdots x_2x_1x_0$.

- 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*=0th 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$\underline{1\mathbf{0}}$	1010	3
$-$	1010			
	0110			
RS	0011	011$\underline{01}$		2
$+$	1010			
	1101			
RS	1110	101$\underline{10}$		1
$-$	1010			
	0100			
RS	0010	010$\underline{11}$		0
RS	$\underbrace{0001}$	$\underbrace{0010}$ 1		

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

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

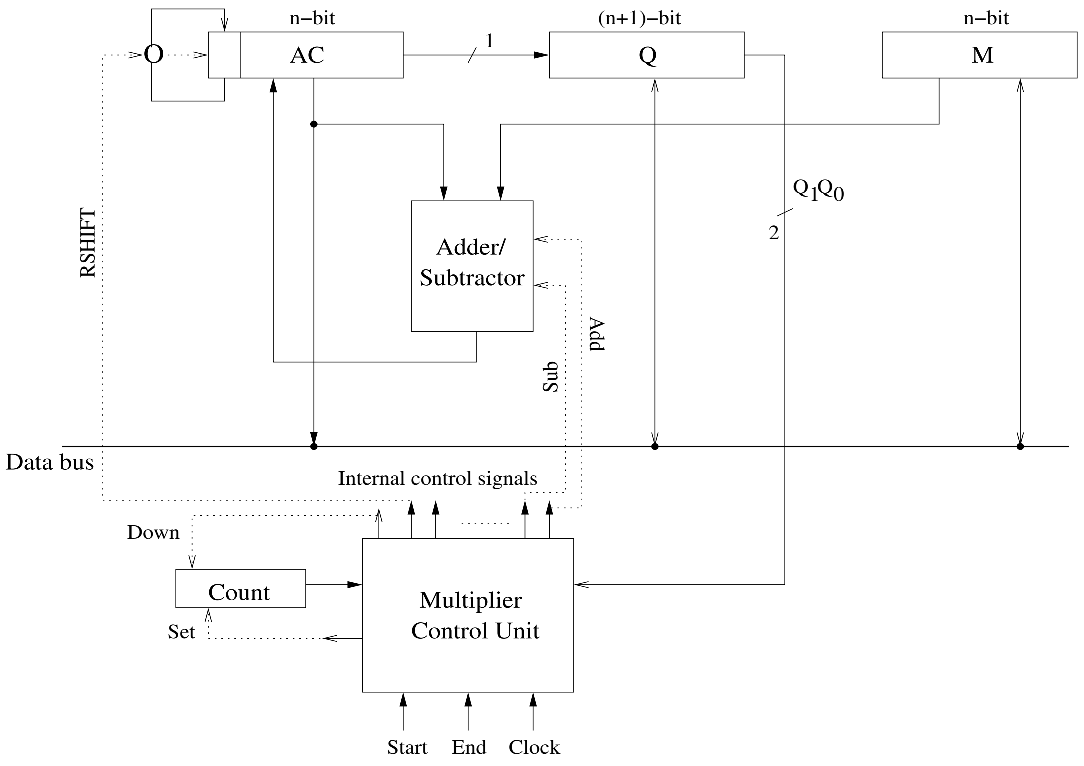


Figure 30: Booth's multiplication algorithm hardware realization

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 agorithm 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 - Multiplier = 01010101

Q - Multiplier with augmented 0 at least significant position = 010101010

Number of 01 pairs in Q (additions in Booth's algorithm) = 4

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,

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

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

Similarly, for 010

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

Hence two arithmatic 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	$PP_{i+1} = PP_{i-1} + 2^iY$
0	1	0	$PP_{i+1} = PP_{i-1} + 2^iY$
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	$PP_{i+1} = PP_{i-1} - 2^iY$
1	1	0	$PP_{i+1} = PP_{i-1} - 2^iY$
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.

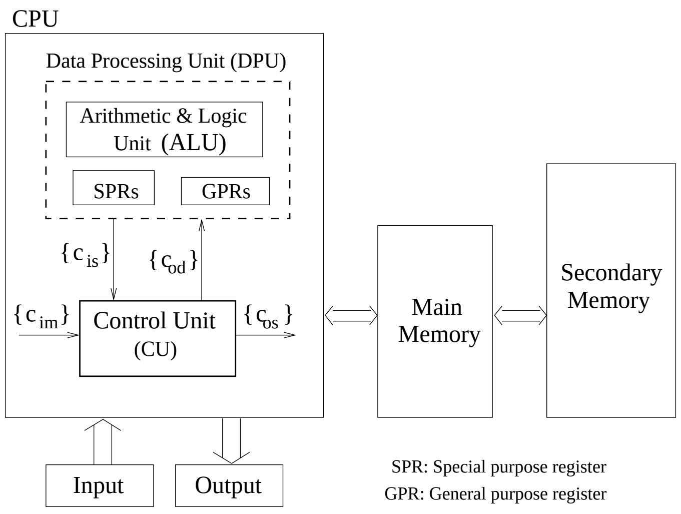
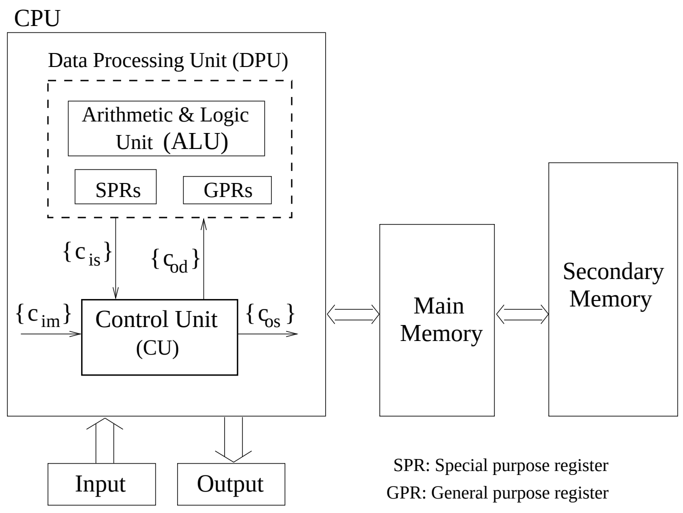

CPU
Data Processing Unit (DPU)
Arithmetic & Logic Unit (ALU)
SPRs
GPRs
{c_is}
{c_od}
{c_im}
Control Unit (CU)
{c_os}
Main Memory
Secondary Memory
Input
Output
SPR: Special purpose register
GPR: General purpose register

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 dignals 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


CPU
Data Processing Unit (DPU)
Arithmetic & Logic Unit (ALU)
SPRs
GPRs
{c_{is}}
{c_{od}}
{c_{im}}
Control Unit (CU)
{c_{os}}
Main Memory
Secondary Memory
Input
Output
SPR: Special purpose register
GPR: General purpose register

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).

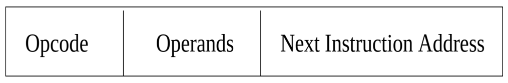


Figure 2: Instruction format containing next instruction address

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 (insruction 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.

$$\mathrm{PC} \leftarrow \mathrm{PC} + 1$$

or

$$\mathrm{PC} \leftarrow \mathrm{PC} + k$$

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,

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

For conditional branch, it is

$$\mathrm{PC} \leftarrow \mathrm{X}$$

if branch condition is true. Otherwise,

$$\mathrm{PC} \leftarrow \mathrm{PC} + 1(k).$$

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 arithmatic/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 hardwared design.

0.2 Hardwired Control Design

Design methodologies considered for harddwired 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 maintainance of circuit become more difficult.

0.2.2 Delay element method

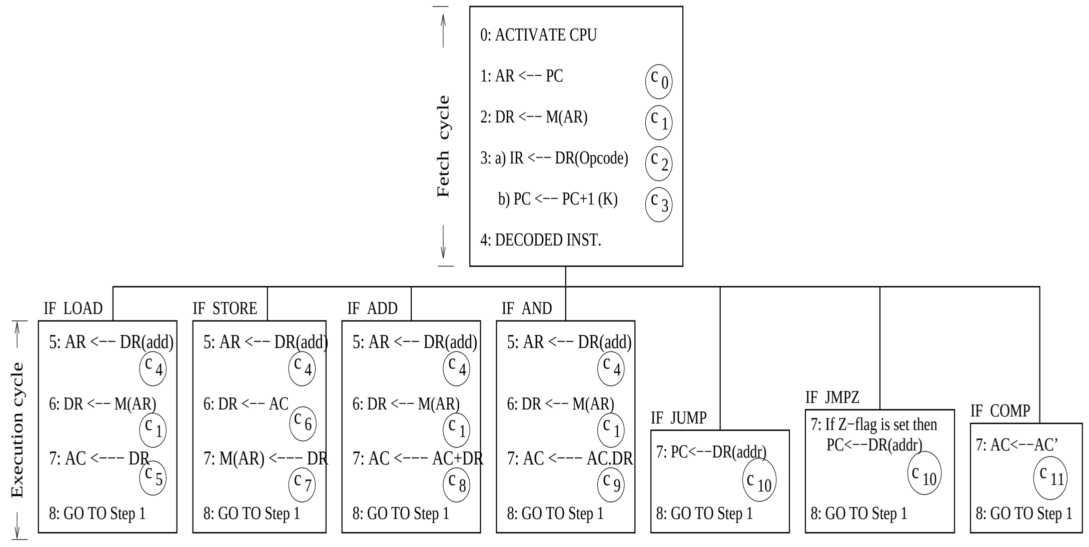


Figure 3: The sequence of control signals

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.

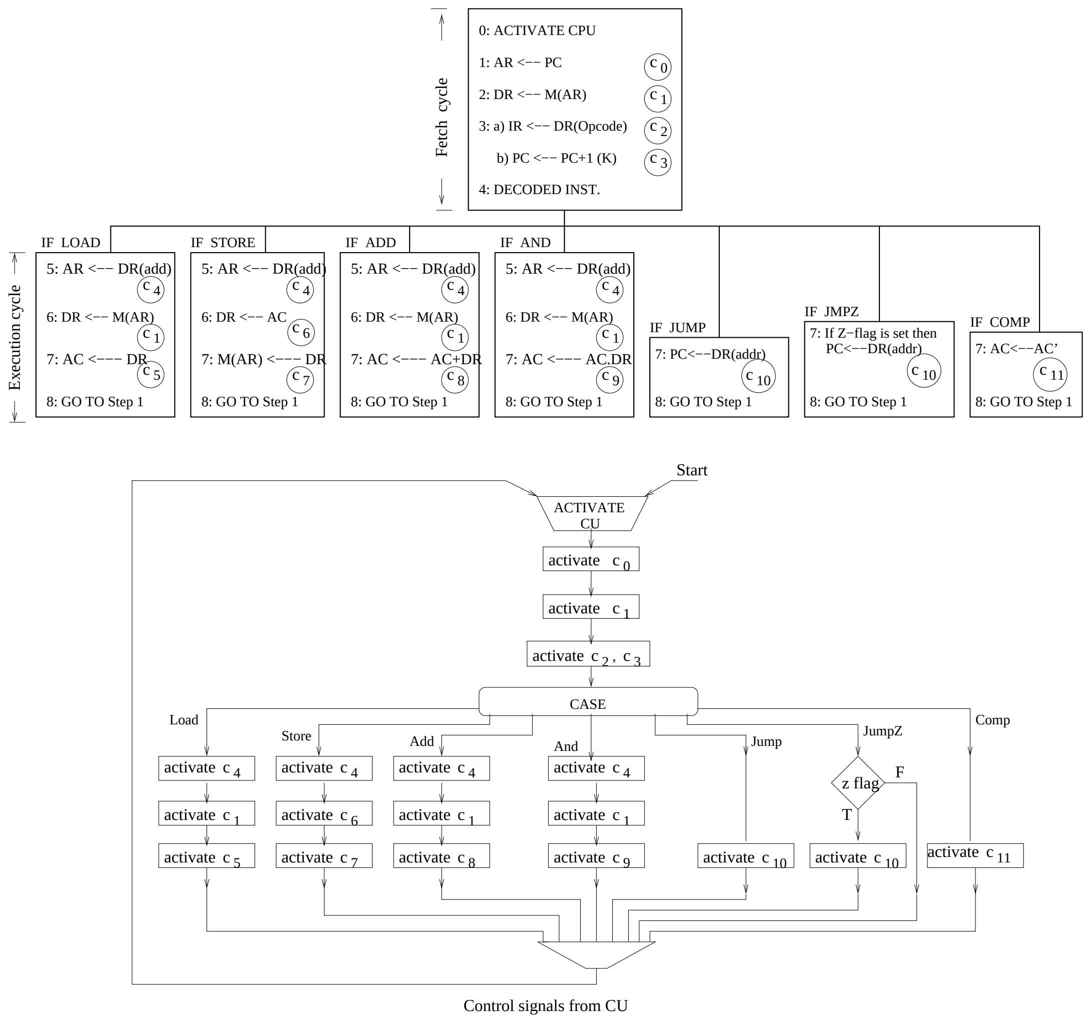
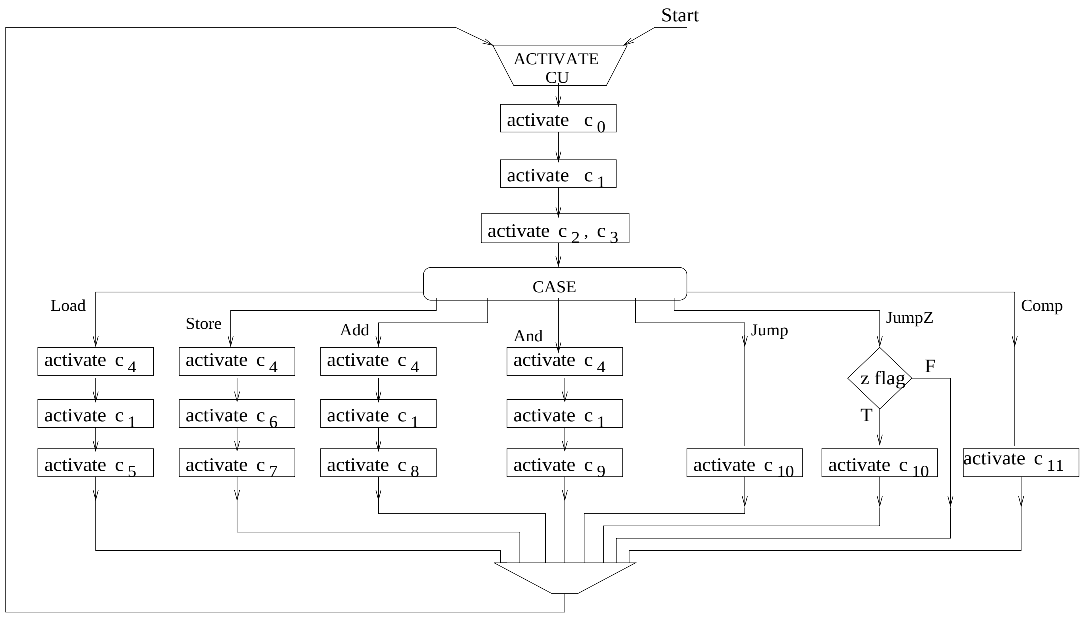


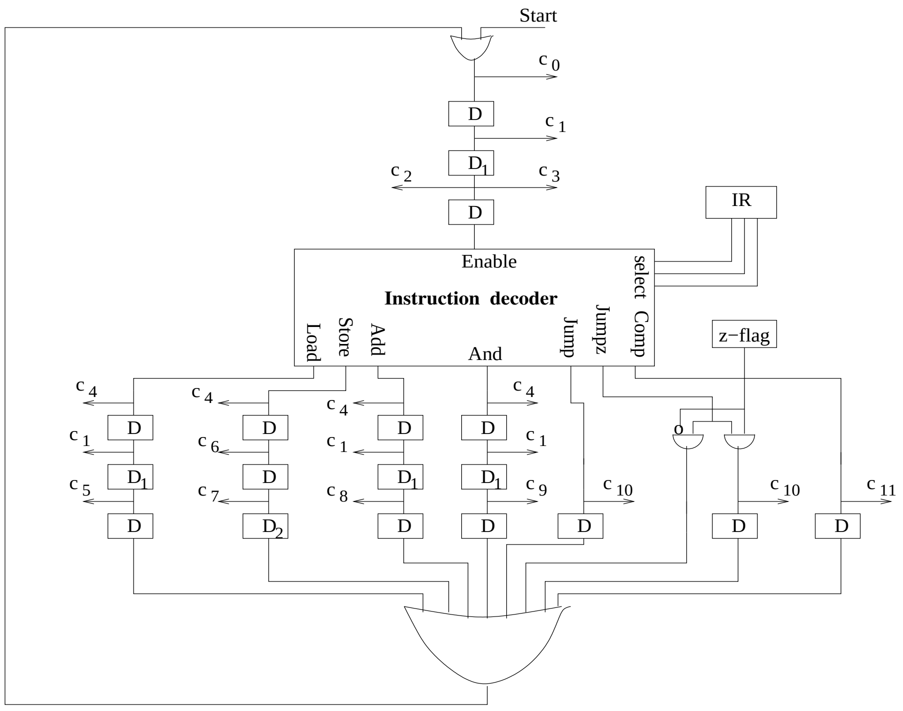
Figure 4: Sequence of control signals

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 dealy 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)).

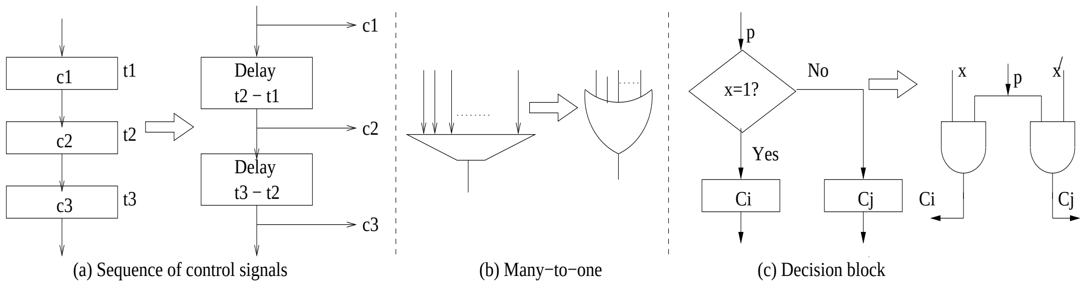


Figure 6: Control flow to logic conversion

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.

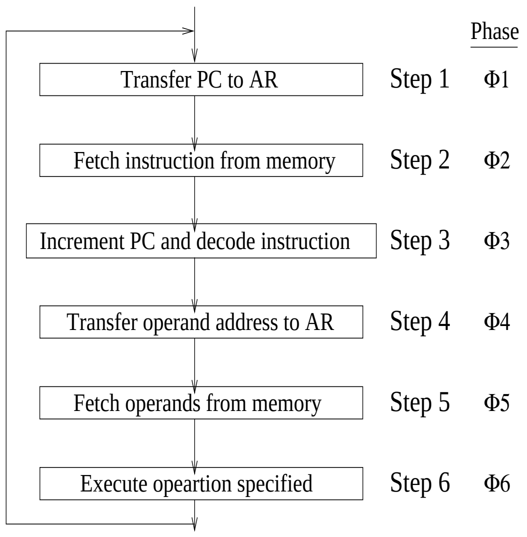

Phase
Transfer PC to AR
Step 1
Φ1
Fetch instruction from memory
Step 2
Φ2
Increment PC and decode instruction
Step 3
Φ3
Transfer operand address to AR
Step 4
Φ4
Fetch operands from memory
Step 5
Φ5
Execute opeartion specified
Step 6
Φ6

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.

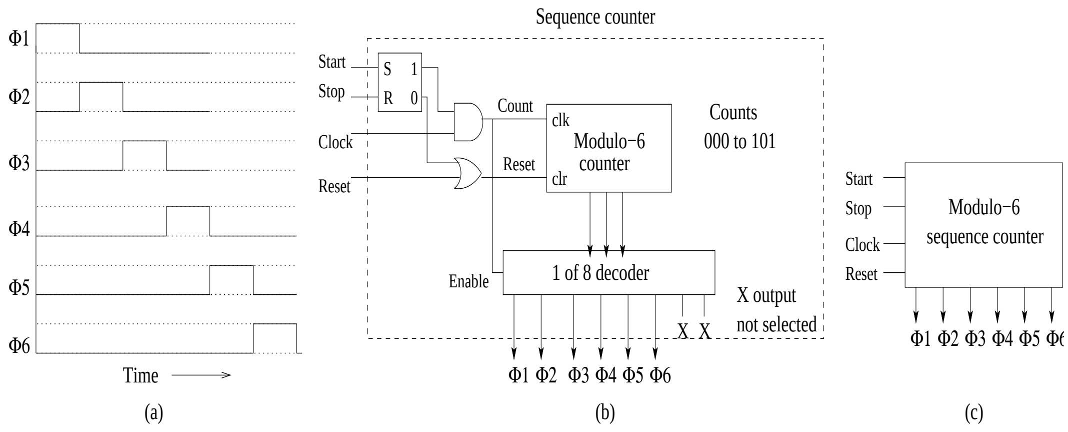


Figure 8: Modulo-6 sequence counter

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_is 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.

Mnemonic	*Description*
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

$Control$ $signal$	$Operation$ $controlled$	
c_0	$AR \leftarrow PC$	
c_1	$DR \leftarrow M(AR)$	$Read\ Memory$
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	$M(AR) \leftarrow DR$	$Write\ Memory$
c_8	$AC \leftarrow AC + DR$	
c_9	$AC \leftarrow AC \wedge DR$	
c_{10}	$PC \leftarrow DR(address)$	
c_{11}	$AC \leftarrow AC'$	$Complement\ Accumulator$

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.

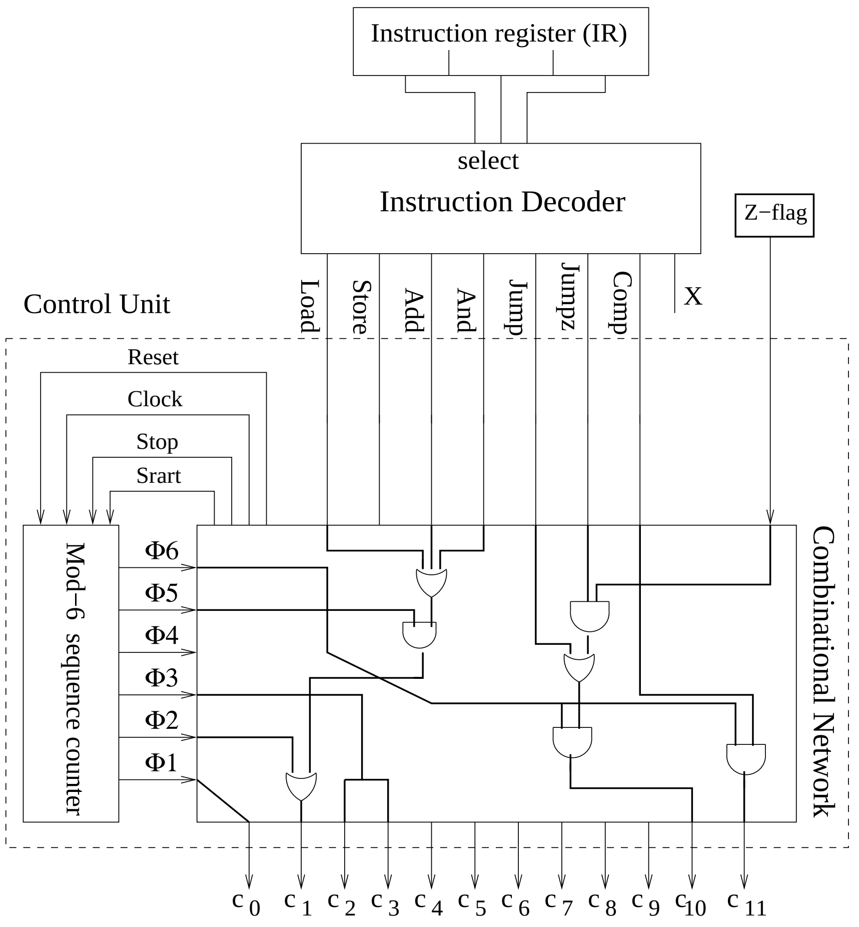


Figure 9: Hardwired CPU control unit around a sequence counter

Example: c_1 which causes a memory read, is activated when Φ_2 =1.

It is also activated when Φ_5 =1 during operand fetching for Load, Add, or And.

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

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

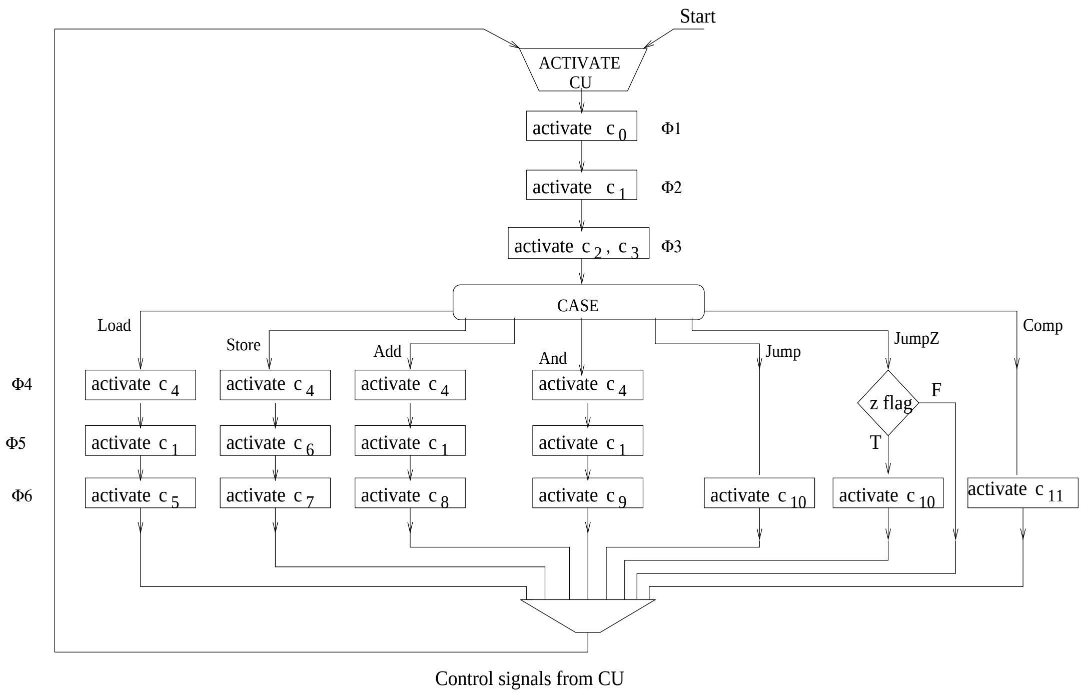


Figure 10: Control signal

The control signal c_i can be defined as:

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

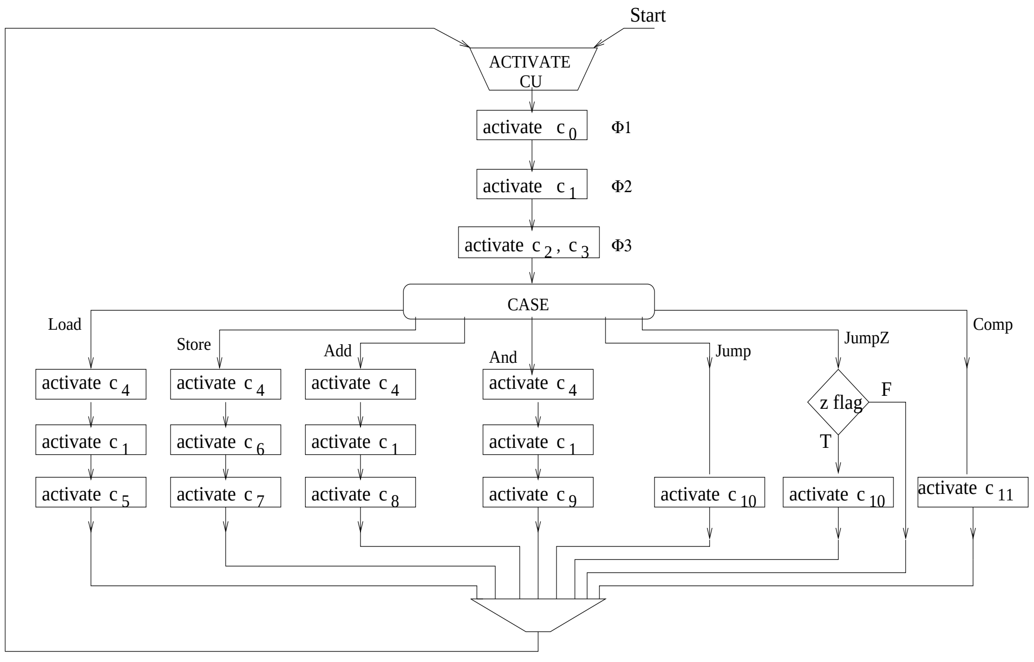
Thus the combinational circuit of Figure 9 is to be such that the c_is are realized.

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

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



Control signals from CU

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

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

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