Basix

05 December 2020 09:02 AM

Computer Types:

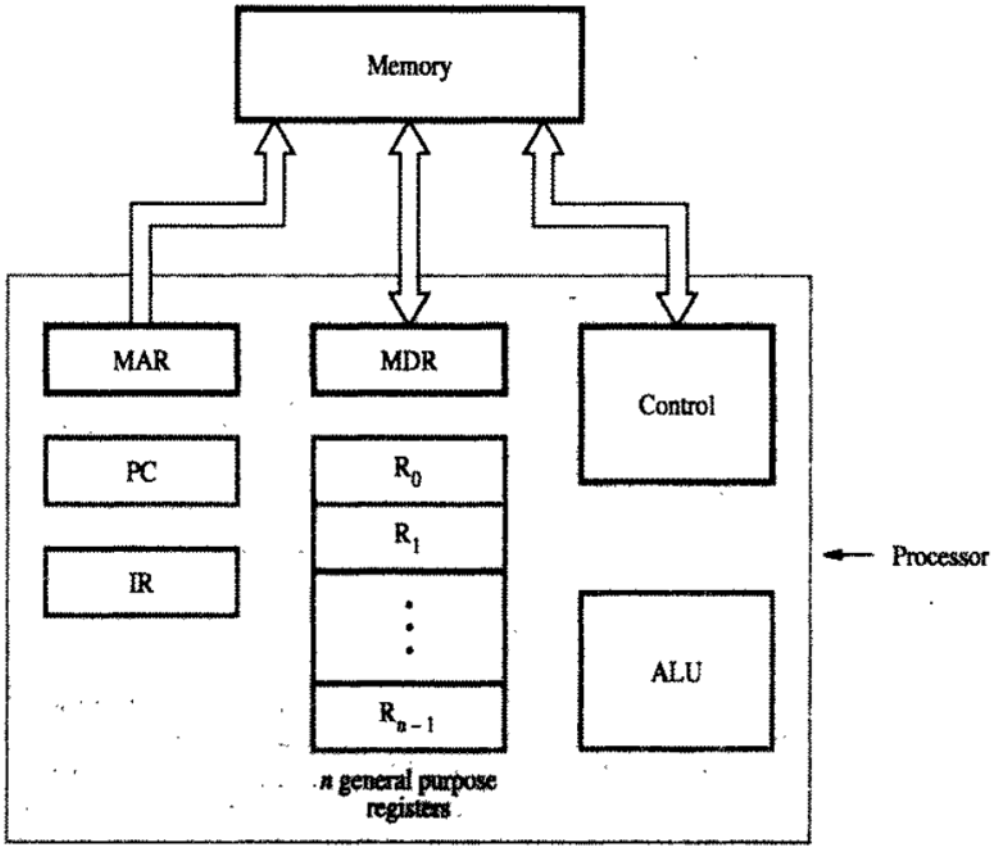
Different types based on processing power:

(desktop computers, Portable notebooks)<Workstations<(Enterprise Computers and Servers)<Super computers

Functional Units:

1. Input : computer receives coded information through input devices.
2. Memory : there are two classes of memory primary and secondary:
   Primary memory are the rams where programs are stored waiting to be executed one by one. These memory contains a large no of semiconductors which stores a single bit of data 0 or 1 a bunch of bits depending on computer architecture are called words and processes operates on these words.
   Primary memory is usually implemented in hierarchies first in the level are caches (fast and small) and at the last level is the main memory(largest and slowest).
   Secondary memory are ssdc and hdds which are cheaper but slower and hence mostly used for Permanent storage , (it cant be used as ram because we need fast read and writes for ram usages)
3. ALU
   this is the processing unit takes some operand data and process it according to some instruction set gives the output data
4. Output : Gives back the output.
5. Control : controls all other components with the help of timing signals.
   (units)

Basic operational concepts:



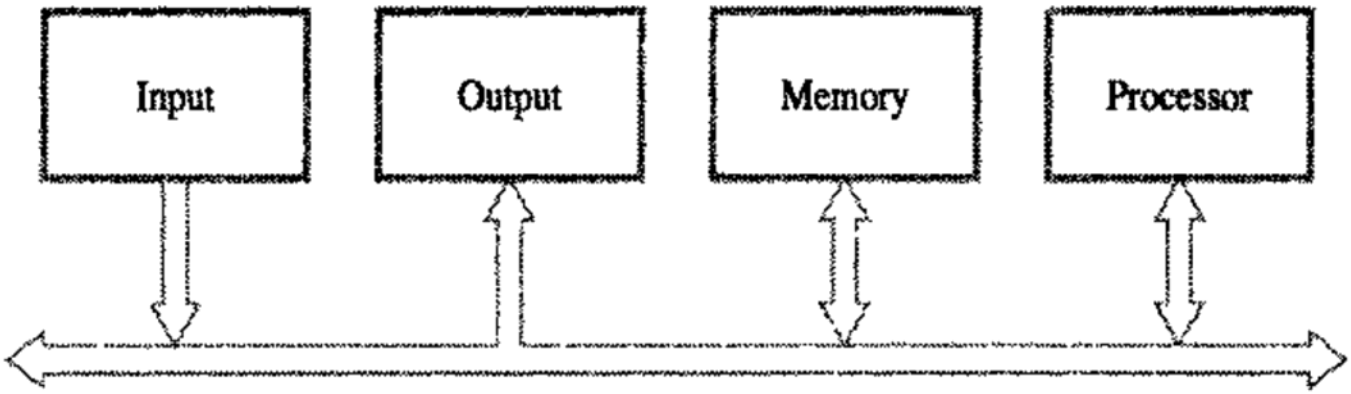
A simplified representation of connection between a processor and memory.

Series of steps involved in single execution of instruction

- Program is compiled and executed and the machine codes are sent to the primary memory (memory).
- PC (program counter) is set to point to the first instruction of the program.
- Contents of PC is transferred to MAR(memory address register) and read control signal is sent to the memory.
- The data (instruction) of the memory cell whose address was stored in MAR is written on to MDR (memory data register).
- After the time required to load data elapses the contents of MDR is transferred to IR (instruction register). No the instruction is ready to be executed.

Bus structure:

Bus is bunch of wires that is used to carry bits from one component to another. For efficient communication between devices efficient connection are required. Most common among them is parallel connection.



In this kind of connection a controller controls the bus and allows only two components to communicate through it.

Software:

- We write program on a program called text editor.
- The text editor runs on another program called operating system.
- The program that we write executes on a particular OS environment.
- The controls of the program shifts back and forth between OS and program to achieve a particular work.

Performance:

Minimizing the time required for operations.
Computer design and architecture affects its speed

Processor clock:

The components of a processor are controlled by a timing signal called clock. The clock defines regular time interval called clock cycles. the processor defines every machine instruction into basic steps each of which it can complete in one clock cycles.
The time length **P** is the time of one clock cycle and the inverse of it i.e. **1/P** is called rate **R**.
More the rate more the processing speed.

Basic performance equation:

If a programme compiles to **N** machine instructions and each machine instruction corresponds to **s** basic computational steps and rate of a processor is **R** then the execution time is given by

$T = \frac{N\,S}{R}$ $\quad \ldots basic\ performance\ equation$

S and **R** depends on the processor also more **R** does not necessarily mean a fast processor it also has to have a small **S**

Pipelining and superscalar Operation

Pipelining : it is the way of achieving a more efficient system and reducing time by reducing S in the above equation, in pipelining we use each component to its full potential . For example in a simple add operation when the content of the memory is transferred to registers or to alu the bus becomes free instead of now waiting for the alu to complete its job we may start executing another instruction since the bus is free we may bring another content of memory to register if so required by the program.

Superscalar operation: it refers to the use of multiple components inside the same cpu so that instruction may run in parallels this reduces the value of s below 1.

Bit representation

03 January 2021 04:27 PM

- An unsigned variable of n bits can hold number between 0 and 2^n-1
- A signed variable of n bits can hold number between -$2^{n-1}$and $2^{n-1}-1$,
  Here one extra bit is used for representing sign.
- Usually the negative are represented as two's complement
- A signed no. -X = an unsigned (complement form) $2^n - x$
- Only negetive no. Are altered to their complements

Complements

I's :

Just invert the bits

2's:

Add 1 to 1's complement

Addition and subtraction in two's complement form

- Addition of two positive integers are as it is.
- Subtraction of two numbers is done as addition of negative and a positive no.
- addition of negative and a positive no.
  - Eg: x-y
  - Y is written in two's complement form as $2^n - y$ $where\ n\ is\ range\ of\ bit$
  - Now x-y => x+($2^n - y$) since -y is written two's complement (if it is 4 bit string -3 will be written as 13)
  - x+($2^n - y$)=$2^n + (x - y$)
  - Now for an n bit string 2^n $will\ be\ written\ as$ 0 $\left(for\ 4\ bit\ string\ 2^4 = 16 = 10000\ the\ highlted\ bit\ is\ lost\right)$
  - x+($2^n - y$)=$2^n + (x - y$)=x-y (numericaly)

Overflow in addition of integers

If a addition result exceeds the limit overflow occurs

More precisely over flow occurs while adding same signed numbers

And can be detected by comparing the sign of result and operands if different overflow has occurred

Memory Location And Addresses

03 January 2021 04:35 PM

Intro

- Memory consists of many millions of storage cells each of which can store a bit of information (1 or 0)
- These cells are usually accessed and retrieved in groups called a word of say n - bits (word length) in a single basic operation
- this word length can typically range from 16 to 64 bits
- A machine instruction can require one or more word for their representation
- A random access to the memory is required for fast operation
- Random access is achieved by having an address to each cell of the memory
- So a address space of k bits is used to refer to a memory location
- Such memory can have up to 2 to the power k addressable location

Byte addressability

- Since a ASCII character requires 8 bits = 1byte of space so usually an address is assigned to every byte of memory

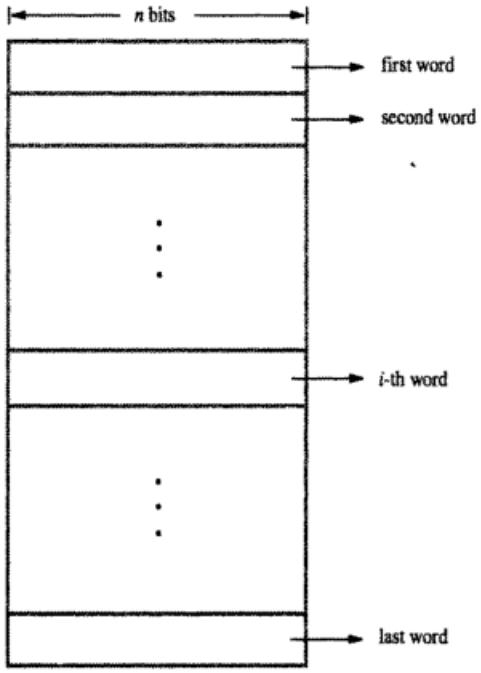


Figure 2.5 Memory words.

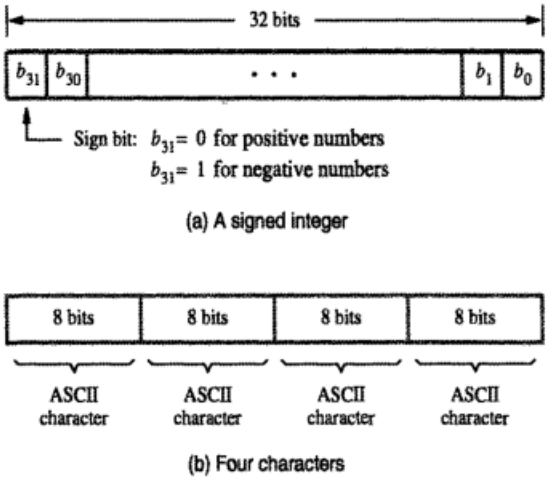


Figure 2.6 Examples of encoded information in a 32-bit word.

- This is usually called byte addressable memory

Accessing numbers characters and strings

Each number or character is stored in a single word so is accessed by just specifying the address to the word

The start of the string is specified by the address of the word and is specified by a special character or a separate memory location can contain the length of the word too

Memory operations and Instructions

05 January 2021 11:43 PM

Memory operations

The two most basic operation between a processor and memory are

Load (read from memory) : to start a load operation the Processor send the address of the desired location to the memory and the memory then read the content of the location and send it to the processor .

Store (write back to it) : The store operation rewrites the content of the memory the processor sends data along with the address where it is to be written and the memory writes the content of that address with the data given .

An information of either one word or one byte can be transferred between memory and processor .

Instructions and instruction sequencing

The computer must have instructions capable of performing 4 types of operations

Data transfers between memory and the processor registers
arithmetic and logic operations on data
program sequencing and control
I/O transfers

Instruction notation

- The actual instruction sets in the processor are not easy to represent while study so we represent it by using certain notations.
- There are two types of notations we generally use assembly language notations and register transfer notation
- register transfer notation
  - Each memory location is given a name eg. LOC,VAR1 etc.
  - Each registers are also given a name eg. R0,R1 etc.
  - Contents of a memory or register are written as [R0],[LOC] etc.
  - So a typical add and store instruction may look like a <- [LOC]+[R0]
- Assembly language notation
  - Each memory location is given a name eg. LOC,VAR1 etc.
  - Each registers are also given a name eg. R0,R1 etc.
  - Here we don't need to specify a notation for contents it is implicitly understood.
  - Eg MOVE LOC,R1 would mean to move the contents of LOC to R1
  - ADD R1,R2,R3 would mean to add R1 and R2 and store the sum to R3

Instruction types

An instruction may contain location of operands or operands and location of memory where the result is to be stored eg: ADD source1,source2,destination

MOVE source, destination

This kind of instruction set requires at most three location and the instruction to be supplied the problem is the size of even one address space + the instruction is overwhelming for a single word So a three-address instruction or two address instruction cannot be used because then we will have to use more words for single instruction that would make the instruction fetching phase of a programme large.

One clever approach is to use two-address space with one address of ram and another of register Since register address space is smaller than ram address space

So ADD source1,source2,destination can be written as

MOVE source1, R1
ADD source2, R1 //this implicitly means add source2 to R1 and store to R1
MOVE R1, destination

Though we are now using 3 primitive instruction but this is faster as the processor fetches an instruction and executes it without waiting for the next part of the instruction to load (fetch is slow because ram is involved). Also if their is one register in processor the register location can be omitted further reducing the size of instruction.

Condition codes

The processor has some special register use to track some status
Called condition code register or status register

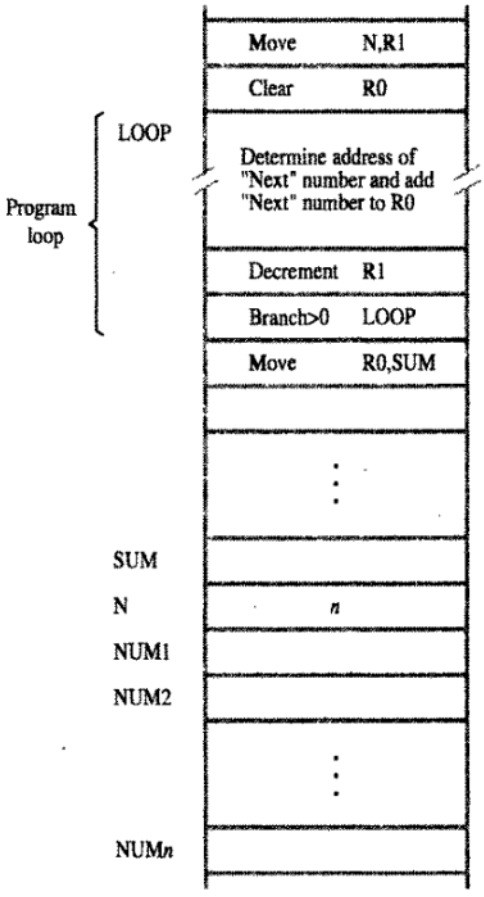
N (negative)	Set to 1 if the result is negative; otherwise, cleared to 0
Z (zero)	Set to 1 if the result is 0; otherwise, cleared to 0
V (overflow)	Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
C (carry)	Set to 1 if a carry-out results from the operation; otherwise, cleared to 0

A typical execution of a programme

Consider the following programme

```
Vector<int> a(10,2);
Int sum=0;
For (i=0;i<10;i++){
sum+=a[i];
}
```

How ram sees it



The main problem now is how to generate address of all 10 numbers inside the loop

This is solved by addressing modes

How processor sees it

1. Move PC (programme counter) to i (start of programme)
2. loads n to R1
3. Set PC to PC + 4 (some where in the middle of execution cycle)
4. Clears R0
5. Set PC to PC + 4 (some where in the middle of execution cycle)
6. Does what is in the loop body
7. Each time decrement R1
8. Check if branch is 0 go to LOOP
9. Else continue after it

Note that branch may be used for if else too.

Addressing Modes

06 January 2021 12:54 AM

Table 2.1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC
Indirect	(R*i*) (LOC)	EA = [R*i*] EA = [LOC]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*]; Increment R*i*
Autodecrement	–(R*i*)	Decrement R*i*; EA = [R*i*]

EA = effective address
Value = a signed number

- When in indirect addressing mode register's address is used it is also called register addressing mode
- Base with index is also called register indexed
- Index is also called register based or displacement addressing
- Base with index and offset is also called register based index

Assembly language

16 January 2021 01:25 PM

Program execution cycle can be written as follows
Source code -> {compiler} -> assembly language -> {assembler} -> machine code(bunch of 0 and 1) -> stored in ram and execution starts.

More info on assembly language in 2.6 carl hamsher

Basic I/O operation

16 January 2021 01:53 PM

Typing something on the keyboard is slower then displaying the same thing on the monitor
displaying something on the monitor is slower then send operation of the processor
Thus we require buffer memory in order to synchronize the input output processes
Communication between two devices of different speeds
Slower device(input/output) records the data in its register (buffer register) and sets a special bit (SIN/SOUT) ->
The processor continuously monitors the (SIN/SOUT) if it is set it copies the contents of buffer register to one of its internal register

(this is very basic more on a dedicated chapter)

Subroutines

16 January 2021 03:18 PM

Certain programmes are already stored in memory owing to their frequent use these are called subroutines
Eg the add operation
When the main programme calls the subroutine the address next to the calling instruction is stored in link register
And pc is pointed to the memory where the subroutine is present at the end of the subroutine the pc is pointed back to the address stored in link register

Subroutine nesting
The problem occurs when a subroutine calls another subroutine in this case the link register will be over written and to avoid so a stack called a processor stack is used a special register called stack pointer is used to keep track of the stack every time a subroutine is called the link address (address next to calling instruction) is pushed onto the stack and every time a subroutine returns control the top link address is popped into PC

Flow of control in subroutine nesting (parameter passing ,stack and stack frame)
Consider the following scenario

Main() -> {calls} -> SUB1 (subroutine1)

1. Main pushes the 4 parameters required by SUB1 onto the processor stack
2. Main calls SUB1 resulting in the return address being pushed onto the processor stack (SP stack pointer now points to this return address).
3. Before the first statement of SUB1 is executed the contents of the frame pointer register FP from main is pushed on top of stack and FP is pointed to this location (where contents of previous FB is now stored) SP and FP both points to this location
4. All the local variable used by the subroutine is now pushed onto the stack
5. All the contents of the register to be used by this subroutine are the pushed onto the stack

6.

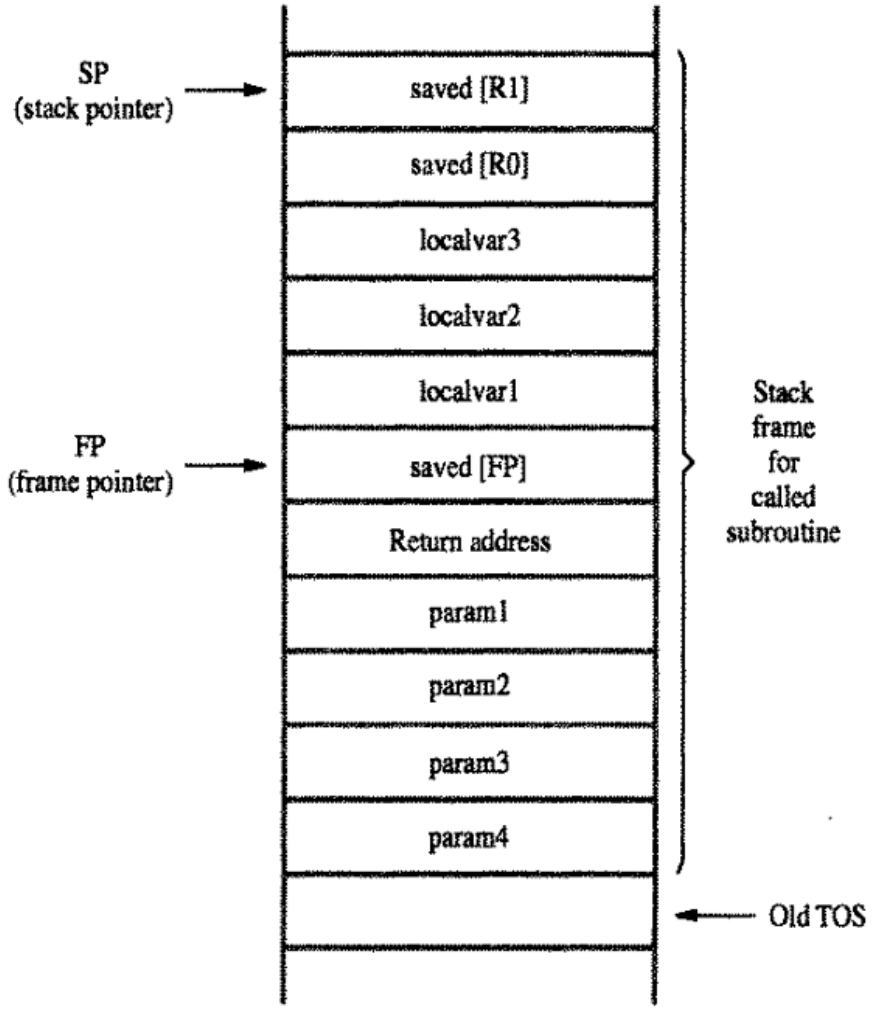


Figure 2.27 A subroutine stack frame example.

This is how it now looks.
At this point the sub routine access the parameters by 8(FP), 12(FP) ....
The local variables are accessed by -4(FP),-8(FP),-12(FP)....

7. After the subroutine is done its task the sub routine pops back the register values (R1 and R0 here) to the specific registers clears its local variables and pops the saved FP back to FP register
8. Which leads to stack pointer pointing on the return address.
9. Controlled is transferred back to return address (to main)
10. Now main clears all the parameters from stack and saves any result from the subroutine (which would by pass back the subroutine here)
11. And now the SP is back to where it started.

More complex example and details on carl hamsher 2.9

Concluding remarks (RISC and CISC)

17 January 2021 12:27 PM

We have discussed how instruction sets work in the processor and we have seen certain instruction set whose OP code would require more than a word this is usually because of sizes of ram address and integers the advantages of this system is we need fewer instruction to perform a particular task -- this is called CISC complex instructions set computing

On the other hand we can also force our OP codes to fit in a word in such a scenario we require every operand in terms of processor register because the addresses are small in that case -- this is called RISC reduced instruction set computing

Section 2.10 and 2.11 of carl hamsher have not been noted it basically provides machine level instruction to some complex programme good for a read

Accessing I/O devices

18 January 2021 07:15 PM

A simple approach to connect all i/o devices is as follows

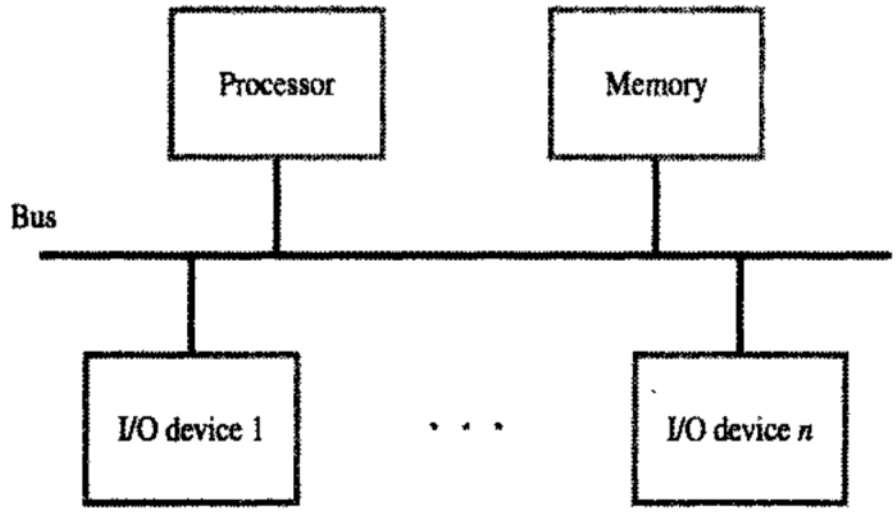
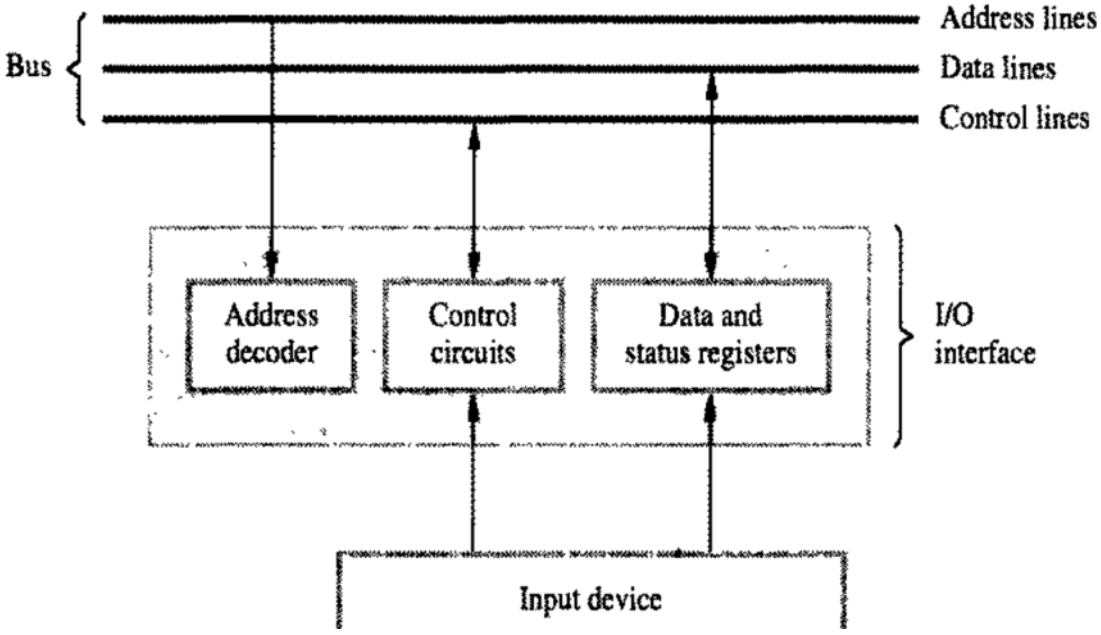


Figure 4.1 A single-bus structure.

- This involves a single bus connecting all the devices directly along with memory and processors
- This bus typically consist of 3 lanes for carrying address of i/o ,data ,control signal providing instructions to i/o
- When the processor places address of an i/o to address line the device identifies itself then responds to the command in the control signal.
- The processor either send a read or write request
- And the required data travels through the data lines.
- When the i/o and memory shares the same address space it is called memory-mapped i/o
- In a memory mapped i/o machine instruction can be directly used to read or write data consider the i/o to be any other memory space.
- memory-mapped i/o is more simpler as it makes software simpler
- Advantage of separate address space(non memory-mapped approach) is that the i/o deals with fewer address lines (since low address space)
- A non memory-mapped approach does not necessarily means a physically separate line for i/o



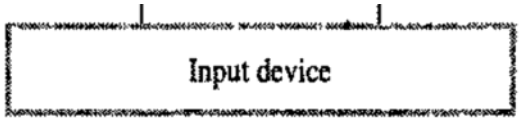


Figure 4.2 I/O interface for an input device.

```
        Move        #LINE,R0          Initialize memory pointer.
WAITK   TestBit     #0,STATUS         Test SIN.
        Branch=0    WAITK             Wait for character to be entered.
        Move        DATAIN,R1         Read character.
WAITD   TestBit     #1,STATUS         Test SOUT.
        Branch=0    WAITD             Wait for display to become ready.
        Move        R1,DATAOUT        Send character to display.
        Move        R1,(R0)+          Store charater and advance pointer.
        Compare     #$0D,R1           Check if Carriage Return.
        Branch≠0    WAITK             If not, get another character.
        Move        #$0A,DATAOUT      Otherwise, send Line Feed.
        Call        PROCESS           Call a subroutine to process the
                                        the input line.
```

Figure 4.4 A program that reads one line from the keyboard, stores it in memory buffer, and echoes it back to the display.

In the above approach the processor continuously checks the buffer for i/o sync called a program controlled i/o. but there are other approaches too.

Interrupts

18 January 2021 10:00 PM

In previous discussion we saw that the program enters a wait loop for i/o to be in sync thus requiring additional processing time from the processor which would have been used for useful computation

In case of Interrupts instead of the processor polling on the input device the input device send an interrupt to the processor.in such a situation at least one of the bus control lines called an interrupt request line is usually dedicated for this purpose.

Let us say we have a job to compute and print certain lines

- the processor runs a compute routine then sends a line to the printer and printer prints the line
- the buffer of the printer can only hold one line for printing
- the processor computes lines one after another and saves it in some memory
- the printer sends an interrupt to the processor
- the processor execute the current statement saves processor state in a stack
- now the processor executes interrupt service routine and sends a line to the printer
- there may be another signal from the processor to the printer which indicates that the interrupted has been Recognised called interrupt acknowledge signal or data transmission may indicate the purpose

Interrupt hardware

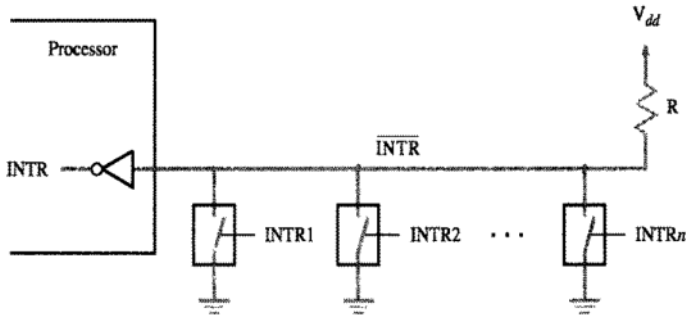


Figure 4.6 An equivalent circuit for an open-drain bus used to implement a common interrupt-request line.

When any of the switch (INTR1,INTR2...) are closed the INTR line voltage drops` to zero and Interrupt signal is activated

Enabling and disabling interrupts

Need for disabling interrupts

Suppose in the previous print statement the processor is not done with computation and a printer interrupt is raised in such a scenario it is desirable to ignore the interrupt.

There are three ways we can disable/enable interrupts

1. Using a machine instruction which clears a bit in processor status(PS) to indicate that interrupts are disabled
   Typically in first statement of interrupt service routine we can disable interrupts
   The statement just before return enables it back
2. Automatically by processor: the processor disables the interrupt just before entering the interrupt
   Service routine and enables it back after returning from it.
3. Edge triggered - the processor receives only one request regardless of for how long there interrupt
   line is triggered for a second request the line has to go down and again come up.

Assuming that interrupts are enabled, the following is a typical scenario:

1. The device raises an interrupt request.
2. The processor interrupts the program currently being executed.

---

3. Interrupts are disabled by changing the control bits in the PS (except in the case of edge-triggered interrupts).
4. The device is informed that its request has been recognized, and in response, it deactivates the interrupt-request signal.
5. The action requested by the interrupt is performed by the interrupt-service routine.
6. Interrupts are enabled and execution of the interrupted program is resumed.

Handling multiple interrupts

Every device which can send interrupts have a bit in the status register called IRQ bit.
For keyboard it may be called KIRQ for display it is called DIRQ etc.
Whenever a device sends an interrupt request the corresponding bit in status register is set

So a simple approach is to let an interrupt service routine poll all these bits the first bit found to be set will be serviced by the processor.
But again here most of the time it may just poll the bits which are not set so is inefficient.

Vectored Interrupts

In this scenario a device requesting an interrupt to the processor identifies itself directly to the processor. Then the processor can directly execute the interrupt service routine for that device.

- For identification the device may send a special code to the processor over the bus.
- The code may contain the starting address of the interrupt service routine to be executed & the rest of the address is supplied by the processor based on the area of its memory where it has stored the interrupt service routine.
    - This also gives the programmer a flexibility to make a branch instruction from that location if multiple interrupt service routines are required.
- The bus may be busy when the device send an interrupt signal so most computer implements a the following
    - Device send interrupt signal INTR.
    - Processor finishes its work and acknowledges the signal turning on INTRA line.
    - Device switches off the INTR signal.

Interrupt Nesting

- Previously we discussed an architecture where when an interrupt service routine(ISR) is being executed no further ISR will be allowed.
- The problem is when an ISR is taking time and another interrupt is important to attend (eg : interrupt received from a clock to update time)
- To cater such scenario we need a priority level coadded inside the processor where when an ISR is executed the priority level of the processor is raised to that device.
- The interrupt of all the device greater than it's current priority is accepted while others are rejected.
- To implement this we need separate interrupt as well as acknowledge lines as shown.

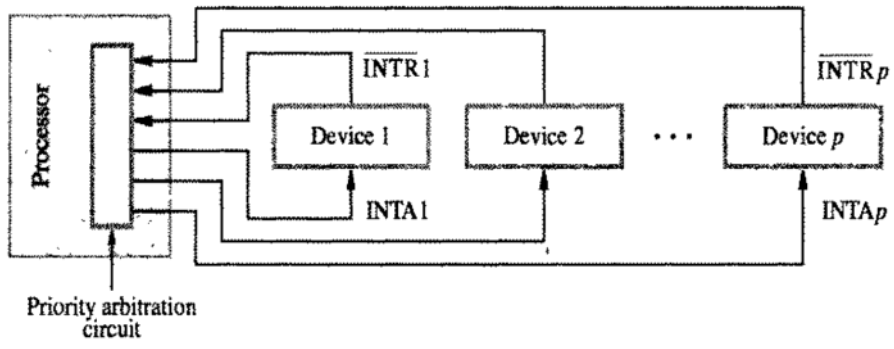
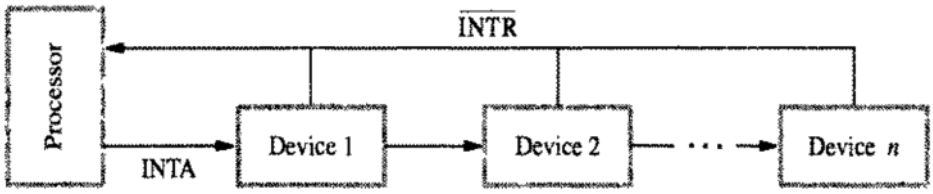


Figure 4.7 Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

The priority level can be encoded by privileged instruction this means no user instruction can change the priority level the privileged instructions are special instruction which are run by the OS itself.

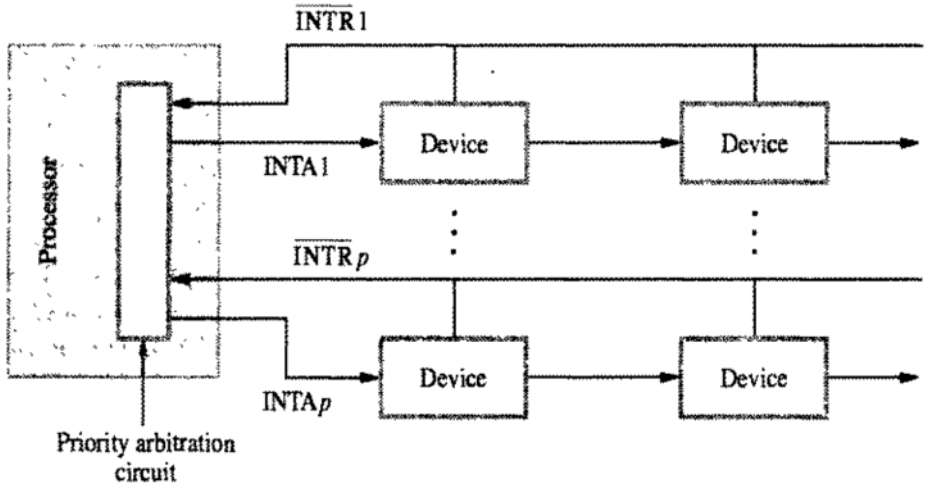
Simultaneous request

- When two or more device interrupt simultaneously one way to choose device to be serviced is through priority structure.
- But when not using priority we can use a daisy chain technique shown below.



(a) Daisy chain

- When the processor receives interrupt through INTR` line it send acknowledge signal through INTA line.
- The devices are arranged in priority
- INTA signal goes to device 1 if it requires to be serviced it stops the signal the places it vectored codes on the data line otherwise it passes the signal for next device.
  A more advanced priority is usually used in many computers as shown



(b) Arrangement of priority groups

Figure 4.8 Interrupt priority schemes.

Controlling device request

28 January 2021 11:02 PM

In all the previous discussions it is assumed that whenever a I/O device is ready (it sets its SIN bit) it sends an interrupt.
But in a scenario where a I/O device is ready but the program is not using it i.e. It is idle in that case it must not send an interrupt.
A special bit is used in the device interface circuit for this purpose called the enable bit KEN for keyboard Den for display etc.
Thus their are two independent mechanism to control the interrupt one at the processor end (special bits in PS register)
Another in the device interface circuit.

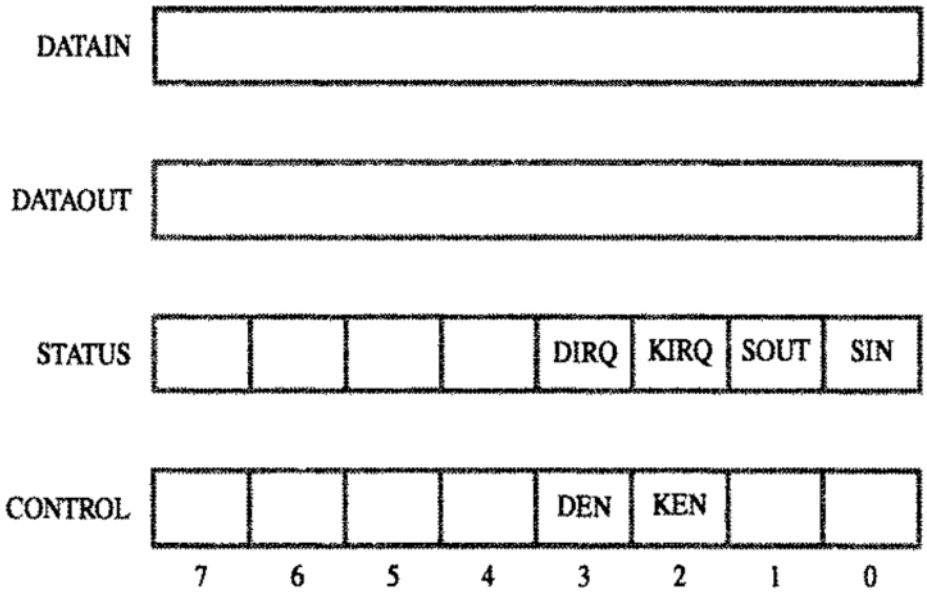


Figure 4.3 Registers in keyboard and display interfaces.

Exceptions

29 January 2021 12:16 AM

An Interrupt is an event where the processor stops the current execution of programme to execute another programme
An event that causes an interruption is called exception.
I/O interruption is one such example of exception.

Here are some other examples of exception.
Recovery from errors

When some error occurs example a hardware error detected by OS or An attempt to divide by zero detected by processor it stops the programme and starts exception service routine.
We saw in case of interrupt service routine the current statement is executed and then the control is transferred but here the processor may immediately transfer controls.

Debugger

The debugger used in IDE uses this exceptions to interrupt the processor and examine the contents of registers and memory.

Privilege exception

When a processor working in user mode attempts to execute instructions of privileged mode.

Some of these exceptions are generate by software (OS or a programme etc.) such interrupts are called software interrupts the OS is responsible for storing the ISR in a specific address of memory and sending the address vector to the processor for same.

Role of OS in raising exceptions -> sec 4.2.6 carl hamsher

Processor examples

29 January 2021 01:40 AM

Skipping for now

Direct memory access

29 January 2021 01:50 AM

To make a transfer from a I/O device to the memory either the processor has to continuously poll the some status register for a single word of bit or some way the whole interrupt mechanism has to work.

For large chunks of data this process has a lot of overheads to reduce these a mechanism called direct memory address or DMA is used.

For DMA operations a control circuit is used with input interface called DMA controller, this controller transfer data without the intervention of processor.

Following registers are present in DMA controller

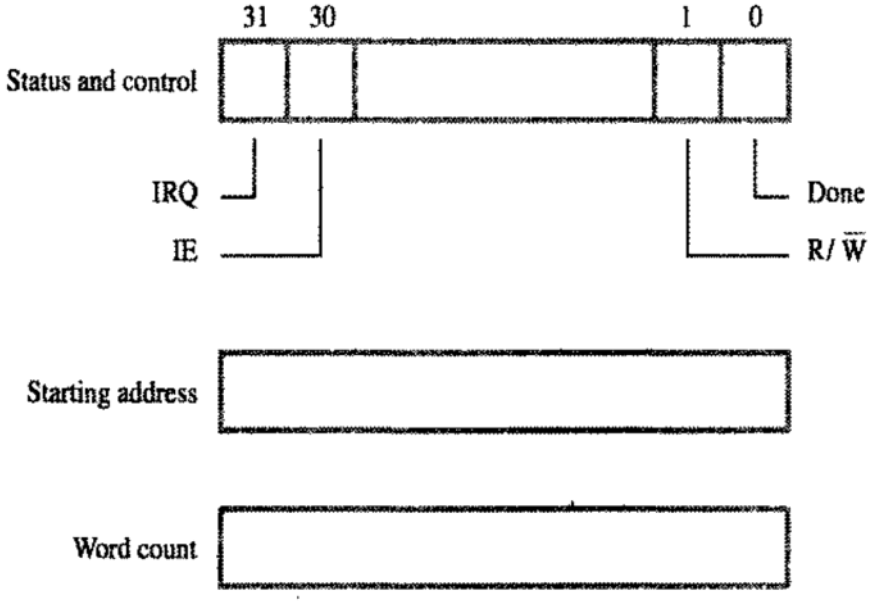


Figure 4.18 Registers in a DMA interface.

The processors sends the starting address and the word count to the DMA controller
The control registers contain informations like

- whether data is to be written or read
- Status register for interrupts
- The work is done or not

The DMA controller connects a high speed network interface to the computer bus
Among the DMA controllers of all the devices top priority for memory access is given to the device with highest speed.

Memory access by the processor and by DMA controllers are interwoven
Since the processor originate most memory accesses the DMA controllers are said to steal memory cycles from processor,
This interweaving technique is called cycle stealing.

When the DMA is given exclusive aces to the memory to do all its work instead of cycle stealing it is called block or burst mode.

For network transfers DMA may also use buffer.

Bus arbitration

29 January 2021 02:32 AM

For a conflict between DMAs and processor an arbitration procedure is used called bud arbitration.

The device that is allowed to use the bus is called a bus master the arbitration technique just chooses who would be the next bus master after the current master is finished.

There are two kind of arbitration techniques
Centralized : in this method the processor or a unit(a single device) connected to the bus will act as an arbiter.

The processor or the unit is the bus master until it grants master ship to one of the DMA
The DMAs are connected to an open-drain circuit in a daisy chain mechanism same way as in interrupts.

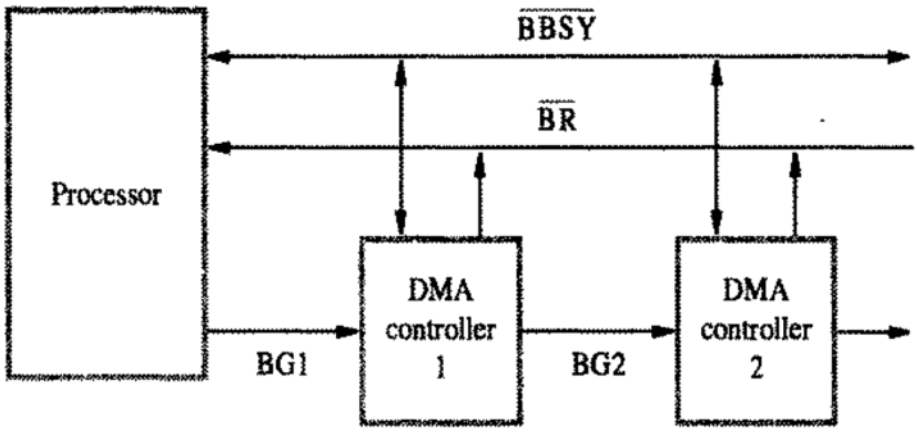


Figure 4.20 A simple arrangement for bus arbitration using a daisy chain.

The DMA sends a signal to the processor that it needs the bus through the BR line.
In tern the processor sends grant signal in daisy chain fashion.
The DMA that activates itself sends a bus busy signal through the BBSY line.
Several such daisy chains can be connected according to priority as in interrupts.

Distributed : in this method all the connected device which is to use the bus will take part in arbitration process.

One such arbitration scheme is as follows

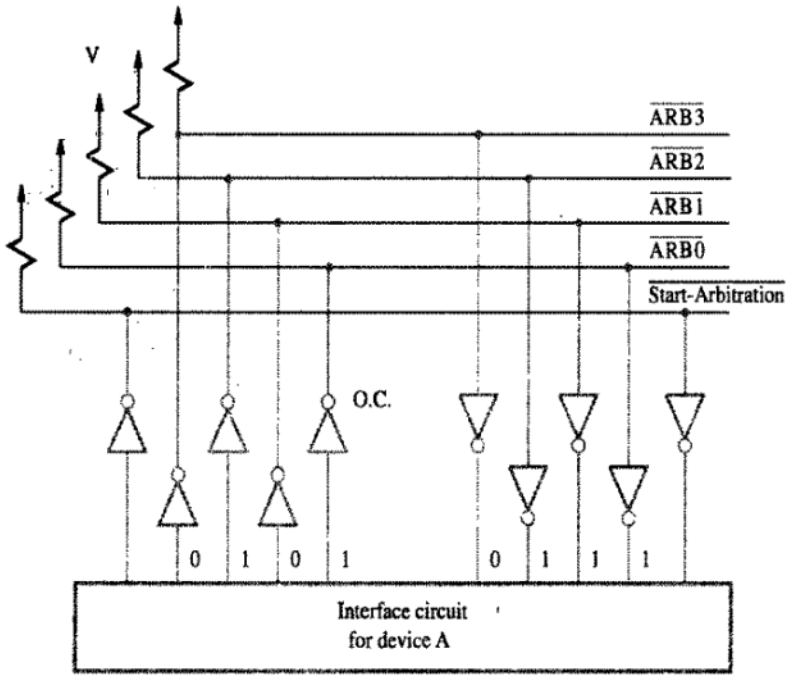


Figure 4.22 A distributed arbitration scheme.

Each of the device is given a 4 bit ID
When devices need to use bus they send a signal on the start arbitration line and put the bits in their ID on these lines
As a result those bits are OR ed
And the result is compared to all the devise starting with their MSB
The one who will have most bits same will win

Buses

29 January 2021 03:23 AM

The processor, main memory and I/O devices may be connected through a common bus which serves as a communication link between them.

The bus lines used for data transfers can be grouped into three groups-
Data - for data transfers
Address - for address to memory
Control - for info such as read or write, timing, data size etc.

Bus protocol : rules that govern the behaviour of device on the bus.

For a particular read write operation two devices are involved one that initiate the request is called mater or initiator and the other is called slave or target.

Broadly there are two types of buses
- Synchronous
- Asynchronous

Synchronous

29 January 2021 02:14 PM

In this bus system All devices derive timing information from a common Clock line.
equally spaced pulses on this line define equal time intervals ,Each interval is called a bus cycle

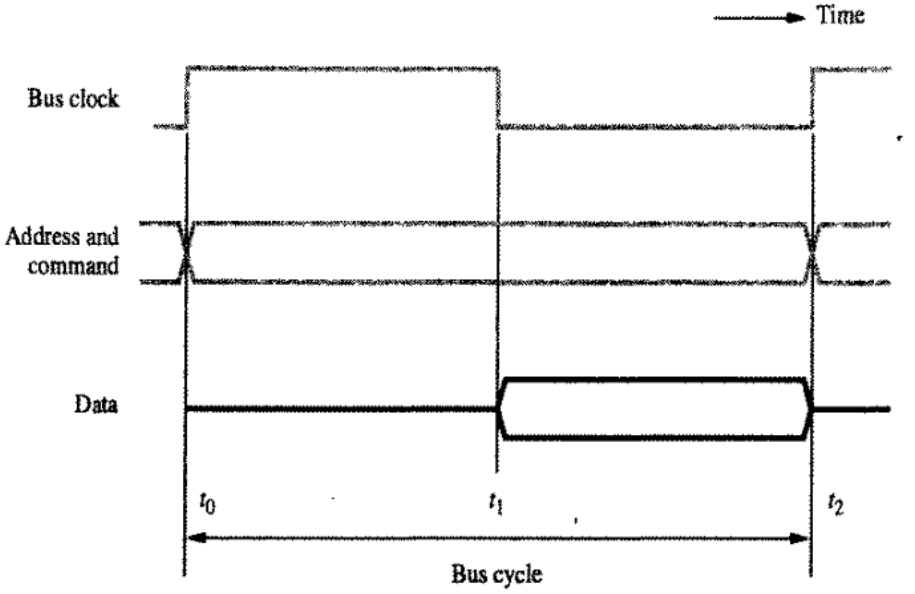


Figure 4.23 Timing of an input transfer on a synchronous bus.

- Between the time To t1 the master generates address and command on the control lines
- Between time t1 to t2 the slave put the required data on data lines.
- at time t2 the master strobe the data from the data line.

however this is an ideal case scenario in reality there is a delay between the master sending the command and the slave receiving it and vice versa ,however the master and slave booth sees the bus Clock identical this is because the designers spend more time to make the bus Clock accurate.

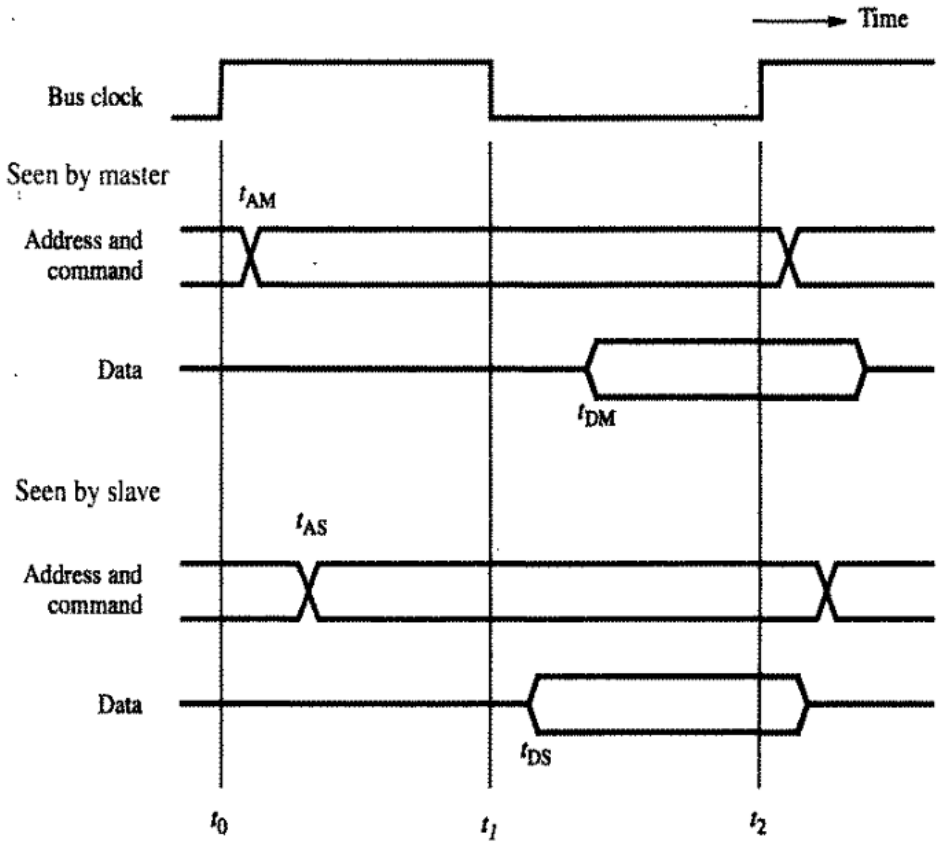


Figure 4.24 A detailed timing diagram for the input transfer of Figure 4.23.

In practical case scenarios however a more complex design is used where by a transfer may take several Clock cycles because the high frequency Clock is used.

also then an address or command is given to the slave the master waits for a slave ready signal and then only it is strobes the data from the data line.

Without the slave ready signal there is no way to make sure that the slave device has received the command without error.

There is a difference between bus frequency and processors Clock frequency
the processor clocks frequency is generally much higher than the bus frequency in synchronous data transfers we are Speaking of bus frequency .

Asynchronous

29 January 2021 02:14 PM

Unlike synchronous bus it does not use any timing signals rather it use a concept of handshake between master and slave using the master ready and slave ready lines.

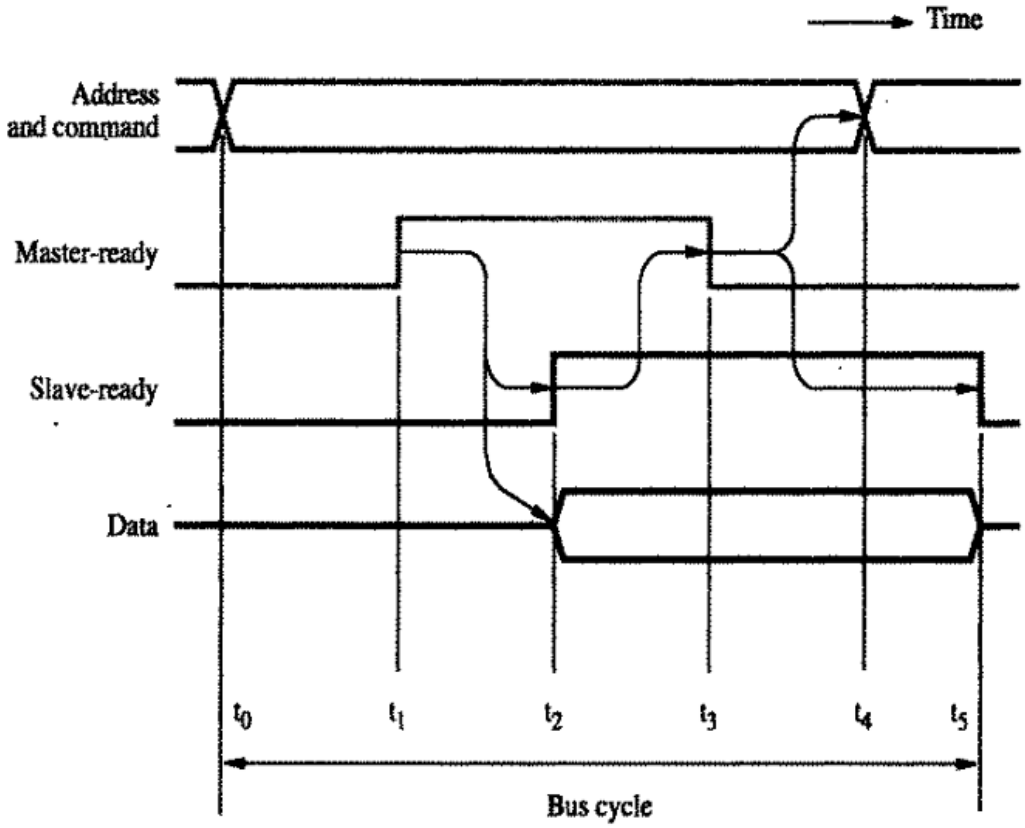


Figure 4.26 Handshake control of data transfer during an input operation.

t_0 — The master places the address and command information on the bus, and all devices on the bus begin to decode this information.

t_1 — The master sets the Master-ready line to 1 to inform the I/O devices that the address and command information is ready. The delay $t_1 - t_0$ is intended to allow for any *skew* that may occur on the bus. Skew occurs when two signals simultaneously transmitted from one source arrive at the destination at different times. This happens because different lines of the bus may have different propagation speeds. Thus, to guarantee that the Master-ready signal does not arrive at any device ahead of the address and command information, the delay $t_1 - t_0$ should be larger than the maximum possible bus skew. (Note that, in the synchronous case, bus skew is accounted for as a part of the maximum propagation delay.) When the address information arrives at any device, it is decoded by the interface circuitry. Sufficient time should be allowed for the interface circuitry to decode the address. The delay needed can be included in the period $t_1 - t_0$.

---

t_2 — The selected slave, having decoded the address and command information, performs the required input operation by placing the data from its data register on the data lines. At the same time, it sets the Slave-ready signal to 1. If extra delays are introduced by the interface circuitry before it places the data on the bus, the slave must delay the Slave-ready signal accordingly. The period $t_2 - t_1$ depends on the distance between the master and the slave and on the delays introduced by the slave's circuitry. It is this variability that gives the bus its asynchronous nature.

t_3 — The Slave-ready signal arrives at the master, indicating that the input data are available on the bus. However, since it was assumed that the device interface transmits the Slave-ready signal at the same time that it places the data on the bus, the master should allow for bus skew. It must also allow for the setup time needed by its input buffer. After a delay equivalent to the maximum bus skew and the minimum setup time, the master strobes the data into its input buffer. At the same time, it drops the Master-ready signal, indicating that it has received the data.

t_4 — The master removes the address and command information from the bus. The delay between t_3 and t_4 is again intended to allow for bus skew. Erroneous addressing may take place if the address, as seen by some device on the bus, starts to change while the Master-ready signal is still equal to 1.

t_5 — When the device interface receives the 1 to 0 transition of the Master-ready signal, it removes the data and the Slave-ready signal from the bus. This completes the input transfer.

Interface circuits

29 January 2021 02:48 PM

An I/O interface consists of the circuitry required to connect an I/O device to a computer bus .
on one side of the interface we have the bus signal for address data and control on the other side we have a data path with its associated control to transfer the data between the interface and the I/O device.

The part that transfers data between the interface and the I/O device is called port.
These ports are of two types

- serial port - Transmits data one bit at a time
- parallel port - transmits data 8 to 16 bits at a time

Parallel port is more costly and more faster then the serial port the devices close to processor uses parallel port otherwise a serial port is preferred.

The interface circuit provides the following :

- Storage buffer that can store at least one word of data
- status bits that can be accessed by the processor to know the status of the interface
- Contains the address decoding circuitry to know when processor request it
- Generates appropriate timing signal for bus control
- Performs any format conversions required such as serial to parallel or vice versa

A detailed circuit of parallel and serial port is given in carl hamsher which is not noted here

Standard I/O interfaces

04 February 2021 02:46 PM

There can be a several alternative designs of the bus of the computer. This variety means the I/O interface suitable for one computer may not be usable for another.

Typically a computer consists of a printed mother board which houses all the circuitry the processor memory etc.

- The mother board has a processor bus which connects all the devices which requires high speed data transfers. This is called processor bus, which connects processor and memory.
- Due to electrical reasons not many devices can be connected to this processor bus, this is why another bus called the expansion bus is connected to the processor through bridge which supports more devices.
- The bridge translates the signals and protocols from one bus to another, witch also introduces a delay compared to processor bus however the processor sees all the devices connected to the expansion bus as if it were connected to its own bus.
- The design of the processor bus is closely associated with the processor hence cannot be standardized, but the expansion bus can be.
- The three widely used busses are
    - PCI (peripheral component interconnect)
    - SCSI (small computer system interface)
    - USB (universal serial bus)

      SCSI and USB can be used to connect devices both inside and outside the PC

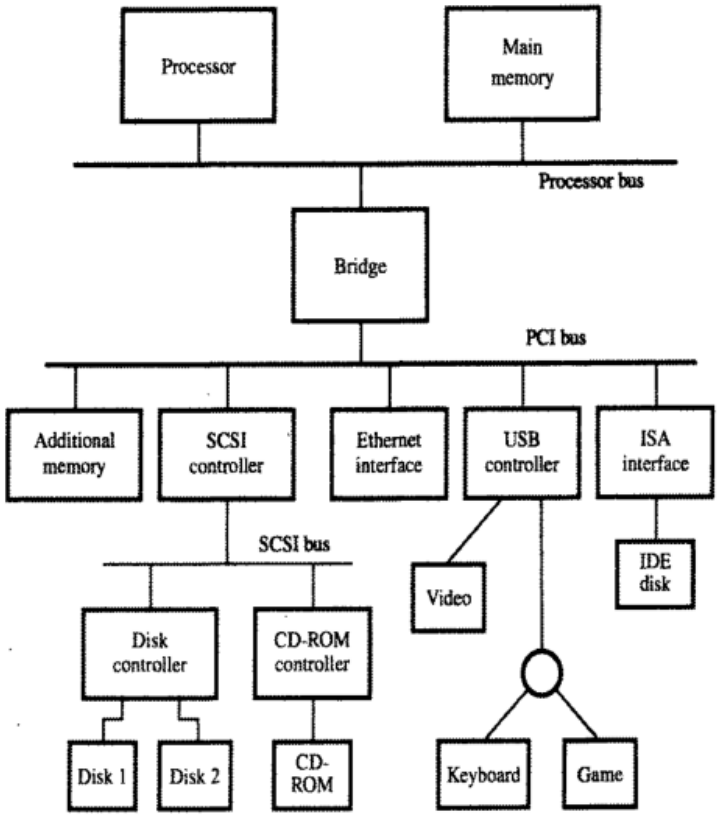


Figure 4.38 An example of a computer system using different interface standards.

A given computer may use multiple buses to give user a range of choices

The individual designs of these individual busses are given in carl hamcher and not noted here

Register names

16 February 2021 09:35 AM

Name	abbreviation	Location	working
Program counter	PC	processor	
Instruction register	IR	provessor	

Fundamentals

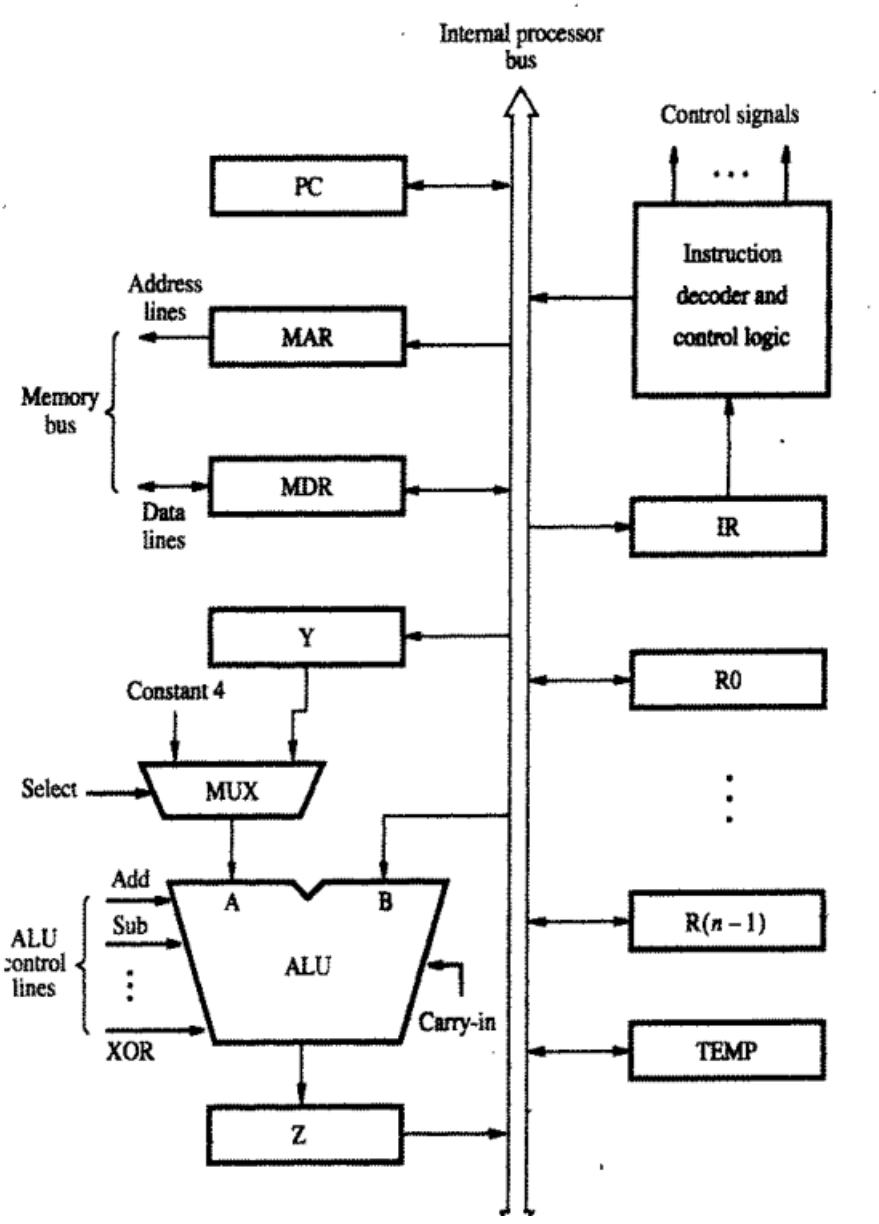
16 February 2021 09:06 AM

The processor is also called ISP (instruction set processor) or CPU (central processing unit) the word 'central' is not evident now a days as modern computers have multiple processors.

Basic steps of working of a processor

1. Fetch : the instruction are fetched from memory and is stored in instruction register (IR) inside a the processor
    a. For instruction having more than one word multiple fetch cycles are performed until the whole instruction is fetched.
    b. PC is incremented in this step.
    c. Symbolically IR <- [[PC]]
2. Decode :
3. Read :
4. Execute :
5. Write :

To study these operations in detail we consider a very simple structure of the processor



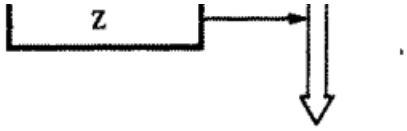


Figure 7.1 Single-bus organization of the datapath inside a processor.

This is a very simplistic design meant only for understanding purposes
Remarks about the organisation:

The control logic block is responsible for issuing signals that controls the sequence of operations in the processor.

The bus that connects the different elements of processor is called internal processor bus.

Almost all the instruction can be performed by performing the following operations in some specified sequence

- Transfer a word of data from one processor register to another or to ALU .
- Perform an arithmetic or logic operation and store it in a processor register.
- Fetch the contents of the given memory location and store them in a register.
- Store a word of data from processor register to a given location.

A detailed discussion of these 4 follows.

Register transfers

16 February 2021 01:23 PM

- Each register is connected to two switches in and out
- The data travel from register Ri to register Rj in the following way
    a. The Ri out and Rj in switch is put to high. (Clk. 1)
    b. Rj loads data in the bus & The Ri out and Rj in switch is put to low. (clk. 2)

The switch is implemented as follows

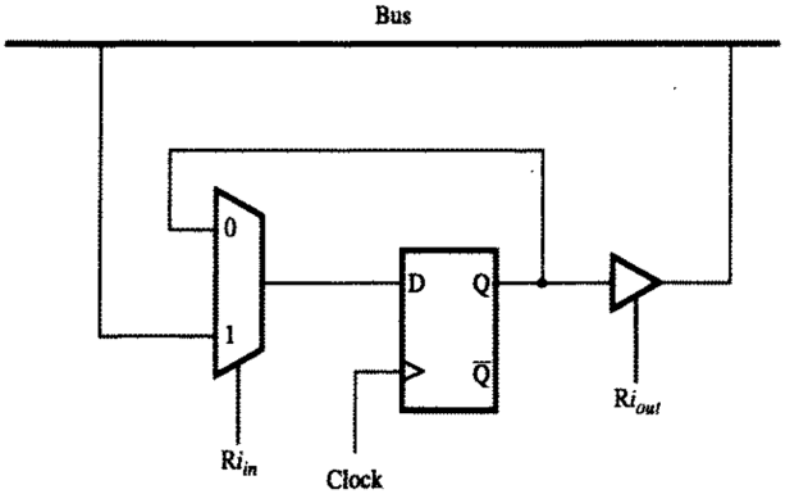


Figure 7.3 Input and output gating for one register bit.

- When R in is high it activates the multiplexer and it selects the data from bus which is stored in flip flop.
- When R out is high the multiplexer feeds back the data in flip flops to bus.

Arithmetic and logic operations

16 February 2021 01:42 PM

The ALU is a combinational circuit which has no storage of its own it just takes two input (one from MUX another from bus) and gives an output which is stored temporarily (in register Z).

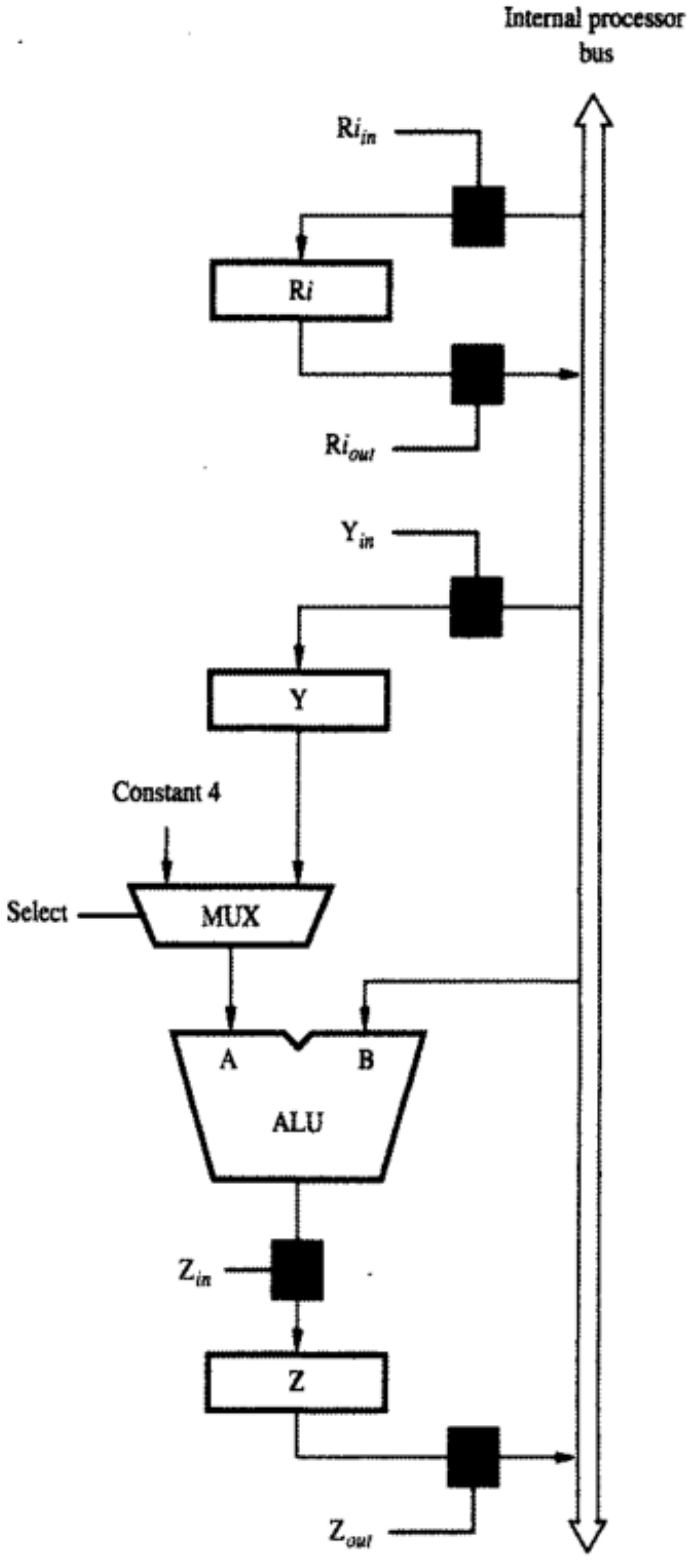


Figure 7.2 Input and output gating for the registers in Figure 7.1.

Example : R3 <- [R1]+[R2]

- R1out, yin (high for Clk. 1)

- R2out, select y, add, zin (high for clk. 2)
- Zout, R3 in (high for clk. 3)

Note. That these discusses the most simplistic design.

Fetching a word from memory

16 February 2021 01:54 PM

The address to be fetched from memory is stored in memory address register MAR
The data retrieved is stored in memory data register MDR
Since MDR communicates to and fro with both internal and external data bus so it is provided with two control gates both controlled by processor.

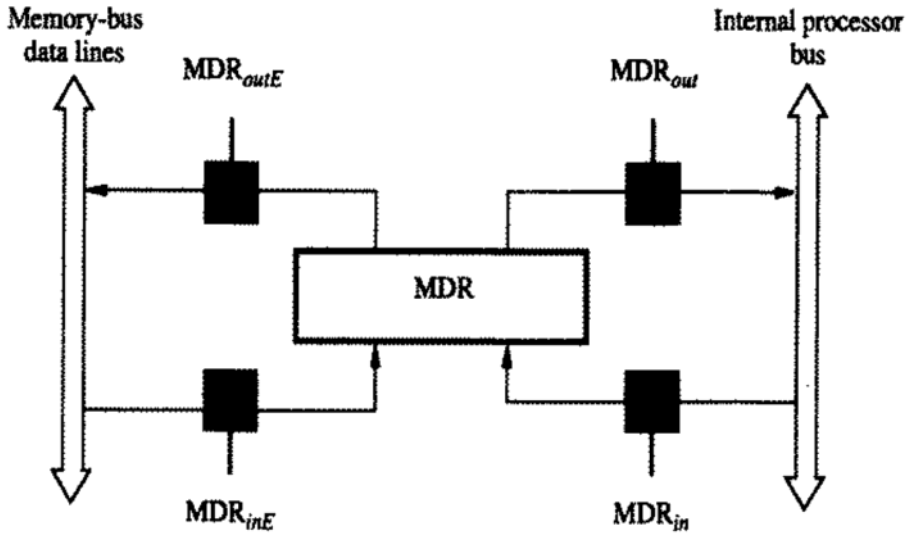


Figure 7.4 Connection and control signals for register MDR.

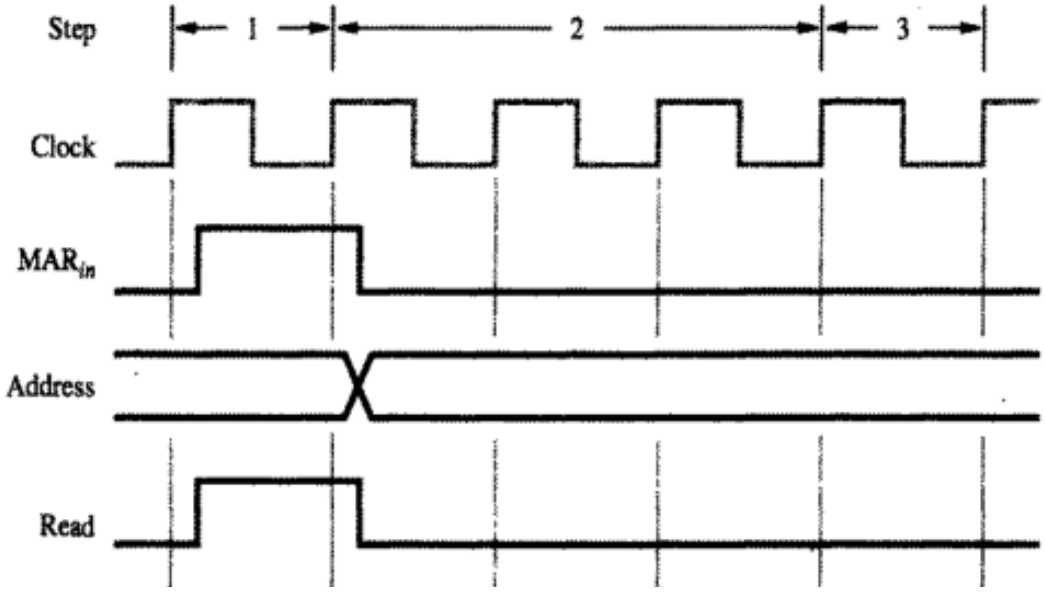
These are the typical steps while fetching the contents of the memory

Consider the operation R2 <- [[R1]]

1. MAR <- [R1]
2. Give a control signal to start a read operation on memory bus
3. Wait for MFC response from the processor
4. Load MDR from the memory bus
5. R2<-[MDR]

In terms of clock cycle

1. R1 out, MAR in, read control (high for clk. 1)
2. MDR external in, wait MFC (high for several clocks from clk. 2)
3. MDR out, R2 in (high for next clk.)



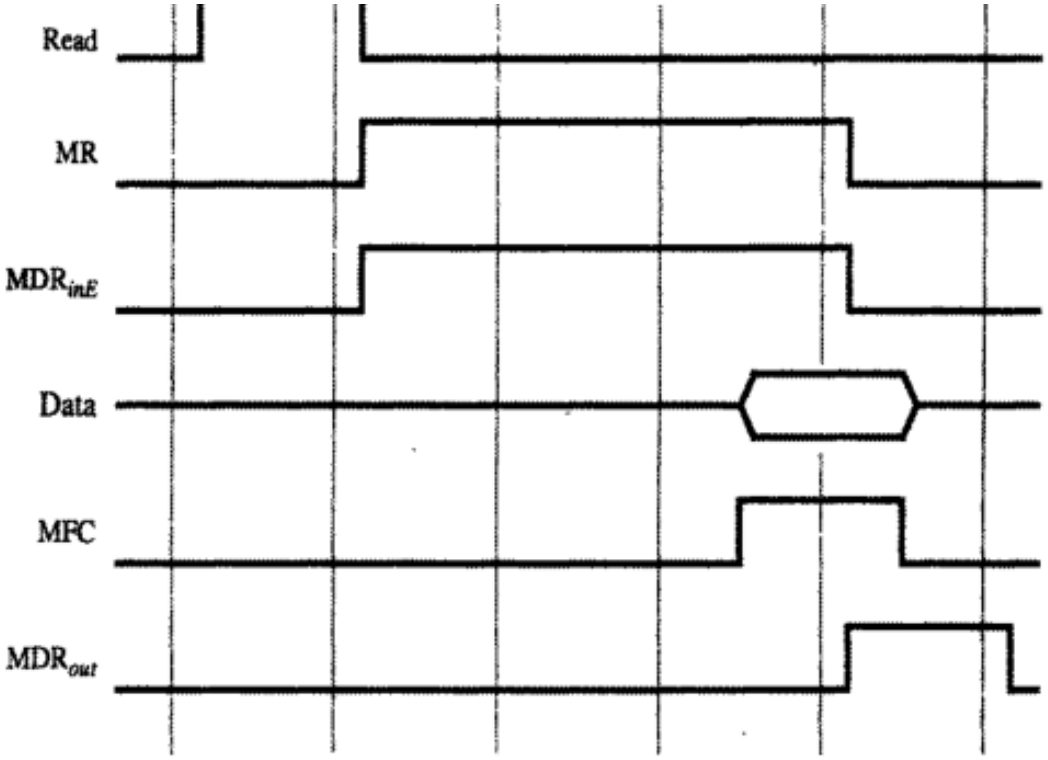


Figure 7.5 Timing of a memory Read operation.

Writing a word to memory

16 February 2021 07:21 PM

It follows almost same as fetching

Consider [R1] <- [R2]

1. R1 out, MAR in,
2. R2 out, MDR in, write
3. MDR external out, wait MFC

Execution of a complete instruction

16 February 2021 07:25 PM

Consider there is a instruction in memory ADD (R3)R1

Following are the control sequence

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R3_{out}$, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

Figure 7.6 Control sequence for execution of the instruction Add (R3),R1.

Explanation

1. Address of PC loaded into MAR; read signal initiated; ALU receives the instruction to increment PC and store to Z register
2. PC is incremented; register Y also holds the value of incremented PC; Wait MFC
3. Transfer instruction from MDR to IR
4. A read operation initiated with address [R3]
5. Register y is loaded with second operand in R1; wait MFC
6. ALU initiated with adding MDR and Y ; result to be stored in Z
7. Transfer from Z to R1
8. END (causes the restart of cycle)

Branch instruction

Consider the following unconditional branch instruction

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Figure 7.7 Control sequence for an unconditional Branch instruction.

Explanation

1. Address transferred from PC to MAR; read signal generated; ALU initiated the task of incrementing PC
2. PC incremented and the value is stored both in PC and register Y; wait MFC
3. Instruction fetched and transfered to instruction register
4. The instruction decoder found a branch, it transfers the offset value to register Z and initiates the ALU add
5. The updated PC which is previous PC + 4 (result of incrementing) + offset is stored in PC

Multiple internal bus arrangements

17 February 2021 08:15 AM

We saw that using a single bus there is a lot of delay as a bus can only be used at a time.

Most commercial processor uses multiple buses to introduce parallel-ity.

Details ommited and in carl hamsher

Control signals

17 February 2021 08:18 AM

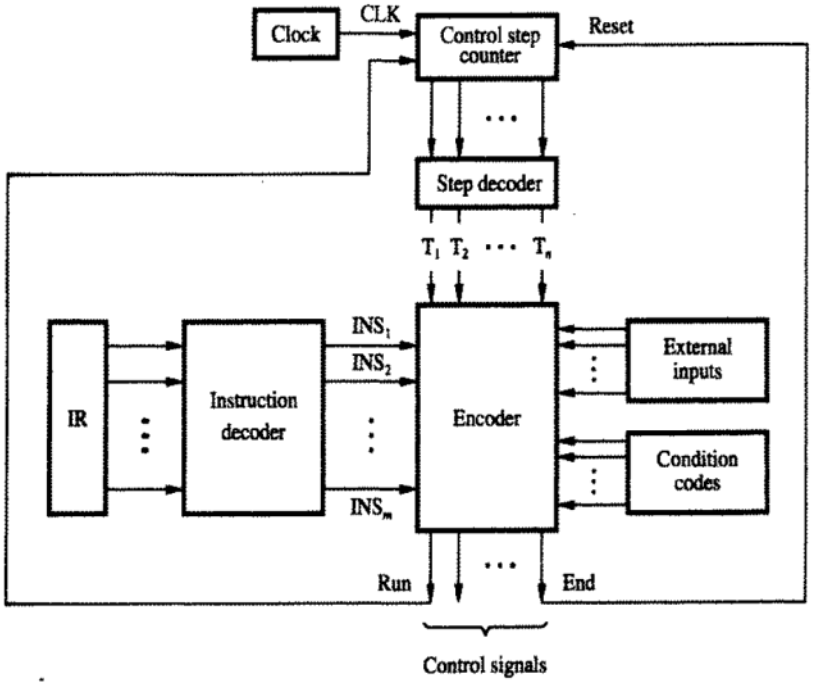
- The execution of commands inside a processor takes place through control signals.
- The control signals must be generated in a sequential order
- To achieve this two types of approaches are used
    - Hardwired control
    - Microprogrammed control

Hardwired control

17 February 2021 08:21 AM

- The control signal that are generated are a function of contents of IR, clock, interrupt signals, condition codes(flags)

The below diagram shows a typical control unit



- Instruction decoder: According to the high bits in IS only of the output lines of the decoder circuit is high at a given time (which corresponds to a particular instruction)
- Step decoder : provides a separate signal for each time slot in a given sequence of instruction.
- External inputs/condition codes : in case interrupts or some error flags these circuit provides the signal.
- Encoder : it activates a bunch of signals depending on it various inputs.

Combining all factors the end result of a particular signal may be

$$Z_{in} = T_1 + T_6 \cdot ADD + T_4 \cdot BR + \cdots$$

$$End = T_7 \cdot ADD + T_5 \cdot BR + (T_5 \cdot N + T_4 \cdot \overline{\overline{N}}) \cdot BRN + \cdots$$

This is called hardwired because at the end every control signal is implemented by some logic gates.

Microprogrammed control

17 February 2021 08:46 AM

- A list of instructions and corresponding signals is stored in a special memory called control store.

Micro - instruction	..	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	:
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

Figure 7.15 An example of microinstructions for Figure 7.6.

- The signals corresponding to micro instruction 1 is [...0111000111000000...] is called a control word (CW)
- This control word initiates what is called a micro instruction or micro routines.
- ??? How does a singe instruction leads to a multiple micro instruction executed

Branch micro instructions (??? When the address generator takes all the status as input why bother specifying branch address in microinstruction if the generator sees a status bit high it may directly generate a signal)

- When a condition flag or interrupt is raised the control circuitry must execute some other micro instructions
- This is accomplished by the specifying in a particular micro instruction the branch address and the bit to be checked to execute that branch address.
- The bit may be a status bit or a bit in IR itself.

To execute aa instruction several micro instruction needs to be executed this is guided by micro program control (μPC)

The μPC is incremented every time a new micro instruction is fetched except in the following situation

- New instruction is loaded in IR μPC is set to start of micro instructions
- Branch micro instruction occurs μPC loaded with branch address
- End micro instruction is occurred μPC is loaded with micro instruction to start the main memory fetch cycle.

Organisation of microprogrammed control

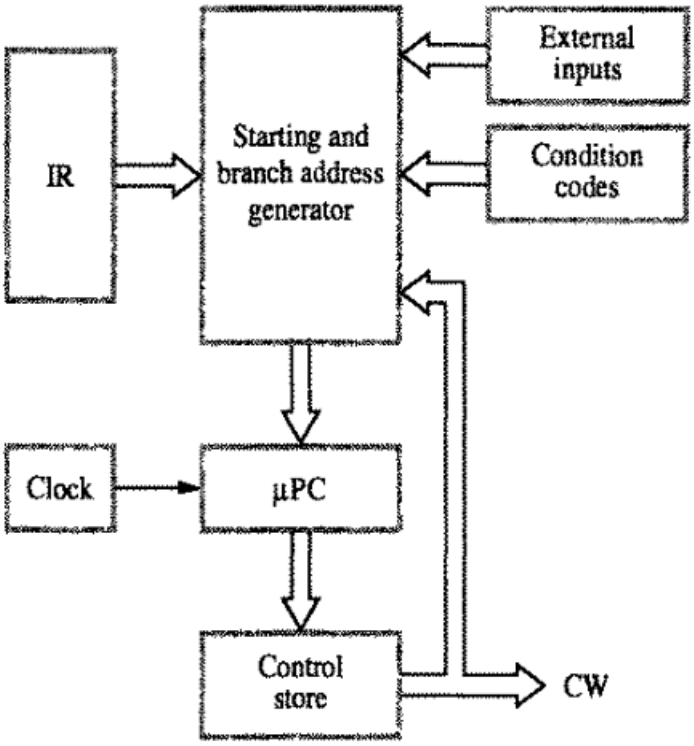


Figure 7.18 Organization of the control unit to allow conditional branching in the microprogram.

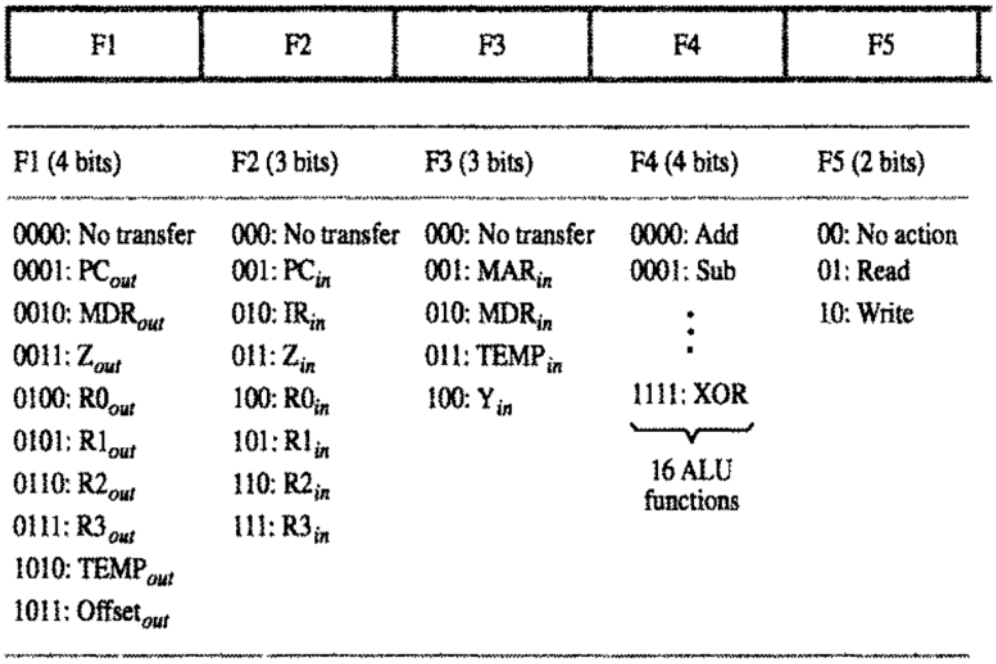
Micro programmes

17 February 2021 03:05 PM

There can be two extremes of micro programmes

- Vertical organisation:

  Here multiple codes are encoded and stored in the control store (reducing the space required in control store), but this needs to be decoded by the decoders (so comes with more hardware complexity)



- Horizontal organisation:

  Here the codes are directly supplied in control store which increases the space required but decreases the hardware complexity.

Micro - instruction	..	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	:
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

Figure 7.15 An example of microinstructions for Figure 7.6.

It is to be noted how ever that this is just the two extremes .

Omissions

17 February 2021 03:27 PM

7.5.2 to 7.5.6

Basix

07 February 2021 03:19 PM

- The maximum size of the memory is determined by the addressing scheme of the computer
  - A 16 bit computer may address up to 2 to the power 16 = 64k locations
  - A 32 bit computer may address up to 2 to the power 32 = 4G(giga) locations
  - A 40 bit computer may address up to 2 to the power 40 = 1T locations
- Usually memory are byte addressable i.e. A minimum of 8 bits = 1 byte can be read from or written to in one go
- A maximum of a word can be transferred between a memory and a processor.
- In a 32 bit computer usually the first 30 bit addresses the word and the next 2 bit addresses the byte in that word

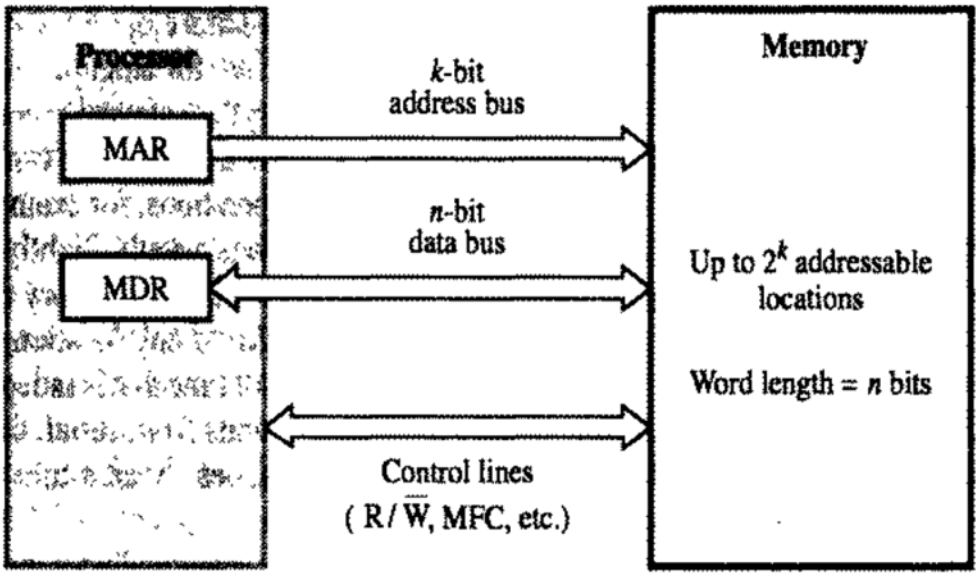


Figure 5.1 Connection of the memory to the processor.

Read write operations in a memory

- An address is placed in the MAR (memory address register)
- If it is a write operation the data is placed in MDR (memory data register)
- R/w control line is set to high if it is read otherwise low
- If it is read operation the memory places the data in data bus and switches on the MFC (memory function complete line) and the processor strobes the data from the bus.
- If it is write operation the memory writes the data on data bus to the given address and switches on the MFC line

Memory access time : the time between the initiation and MFC signal
Memory cycle time : the time between two successive initiations of memory operations

A RAM (random access memory) can access any address randomly independent of address access time is same for all location.

A non -RAM (hard drives etc.) access time depends upon the memory locations

Cache memory

Usually the speed of the processor is much faster than the memory, thus making the memory the bottle neck in design so a cache (smaller, costly, faster) memory is inserted between a much larger RAM and processor. It holds the currently active segments of a program.

Virtual memory

To increase the effective size of the memory an address space as large as the addressing capability of the processor may be used from a hard disk drive. At a time only a portion of this is mapped to actual memory (currently active program) if the processor asks for a location that is in the disk a page of memory is mapped to the physical memory replacing some inactive ones this operation takes more time but making some smart choices may reduce the probability of such operations.

Semiconductor RAM memories

07 February 2021 05:20 PM

Internal organisation

07 February 2021 07:24 PM

- Usually organised in the form of array of bits.

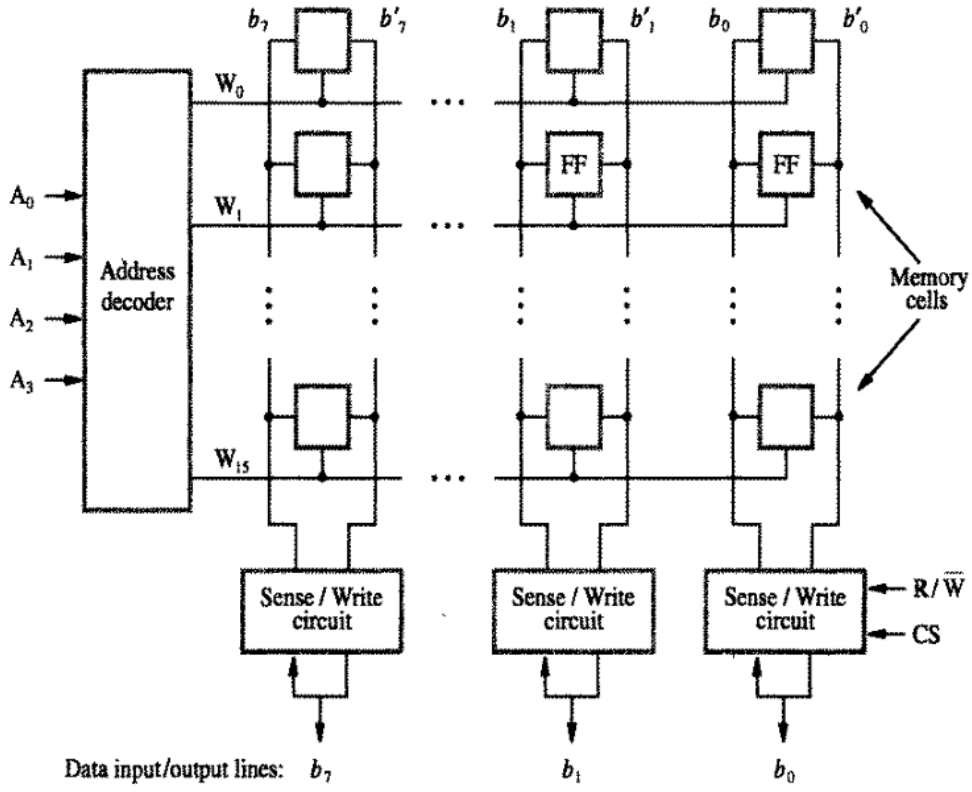


Figure 5.2 Organization of bit cells in a memory chip.

The simplistic chip given above has the capacity of 128 bits organized into rows of words thus we have 16 rows or words each word having 8 bits.
In other words we have 16 rows and 8 columns.
This is called a 16 x 8 organization.
The rows are coupled with a decoder called address decoder.
Each row is fitted with a circuit having one output wire each.
No. of external connections.

Control lines - 2 (R/W,CS)
Power supply - 2 (GND,VCC)
Address lines - 4 ($log_2^{no.of\ rows}/log_2^{16}$ in this case)
Data line - 8 (no of columns / word length)

An alternative approach consider 1024 bit RAM

We may organize it as 1024 x 1 organization

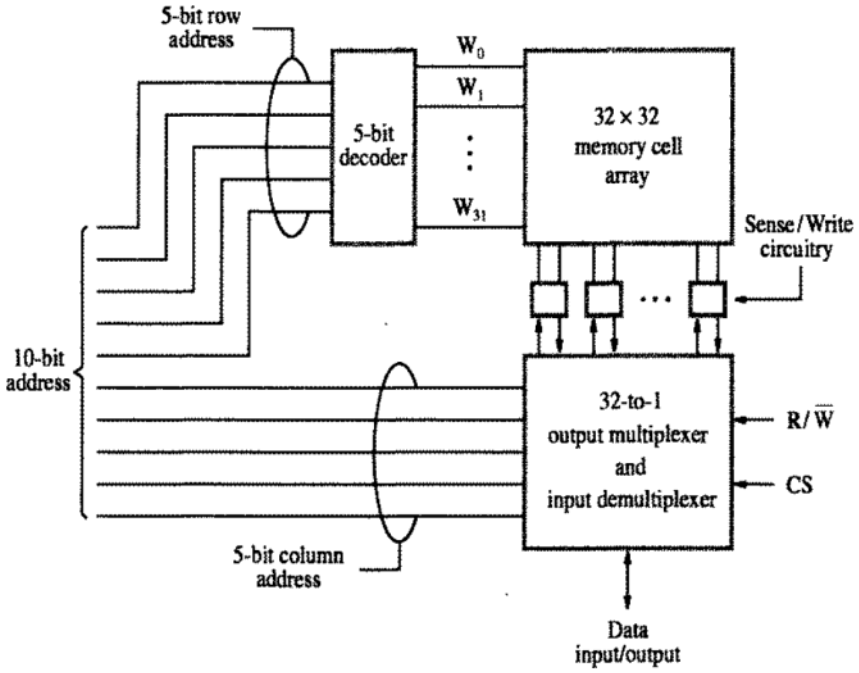


Figure 5.3 Organization of a 1K × 1 memory chip.

To do this we would require log (1024) =10 Address lines and 1 data line
Instead of having a 10 bit decoder we may use the approach given above
The row address selects a row column address selects column and thus a bit is transferred through data lines.

Static memories

09 February 2021 08:51 AM

The memories which require power to store some information and the info is lost as soon as power supply is interrupted is called static memories.

The static RAM (SRAM) uses a latch to implement the single bit storage.
Usually a CMOS latch (using 4 transistors) is preferred because it consumes less power because current actually travels through the circuit while a R/W operation takes place.

The sense/write circuit is responsible for changing and reading the state of those individual latches.

Two implementation of latch using Not gate and CMOS is given in carl hamcher details is omitted here.

DRAMS

09 February 2021 09:11 AM

The SRAMS are fast but comes at a high cost due to a large no. of transistors involved.

In DRAM a much simpler memory cell is used which uses a single transistor and capacitor.
The capacitor holds the digital data but is required to be periodically refreshed hence it is called dynamic RAM.

The details of these cells are given in carl hamsher and is omitted here.

Asynchronous DRAM

A Design of 16 MB DRAM

Memory conversions

1K (kb) = pow(2,10)B (bits)
1MB = pow(2,10) kb = pow(2,20)B
16 MB = 16*pow(2,20)B bits

Now a 16 MB DRAM is organized into 4K x 4K

There are 4k rows 4K columns
The columns are grouped into 512 bytes(8 bits)
Row address lines= log(4k) = log(4) + log(pow(2,10)) =12
Column address lines= log(512) = 9 (byte addressable)

This gives the following 21 bit address organization.

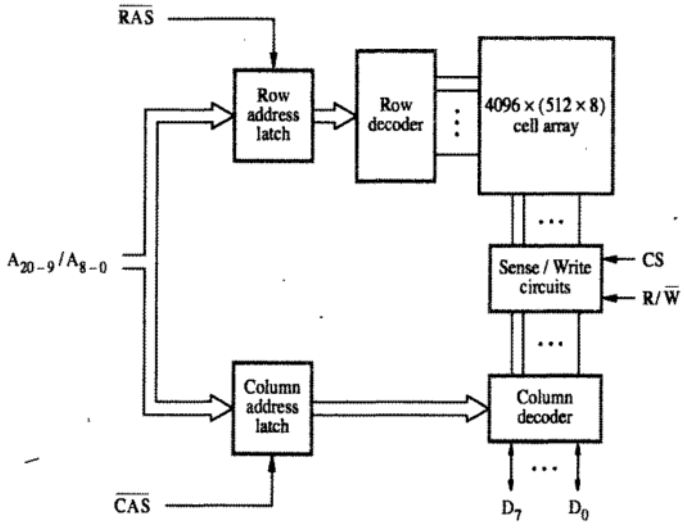


Figure 5.7 Internal organization of a 2M x 8 dynamic memory chip.

External pins are further reduced by multiplexing the row and column address onto 12 pins.
So first row address is applied it is loaded into the row address latch.
Then the column address is applied loaded into column address latch.

In such a memory cell when ever a read operation takes place the information inside the cells are refreshed.

So the first application of row address ensures the refreshing of operations.
The shortly after that column decoder selects a particular cell and then the info is retrieved.

The RAS row address strobe And CAS controls these operations.

These RAMs support fast page transfers also in which all the columns are provided with latch so when a row address is signal is applied all the contents of the row is transferred to these latch and then the CAS control signal can strobe the one at a time, thus

using only those latches and reducing time.

Synchronous DRAMs

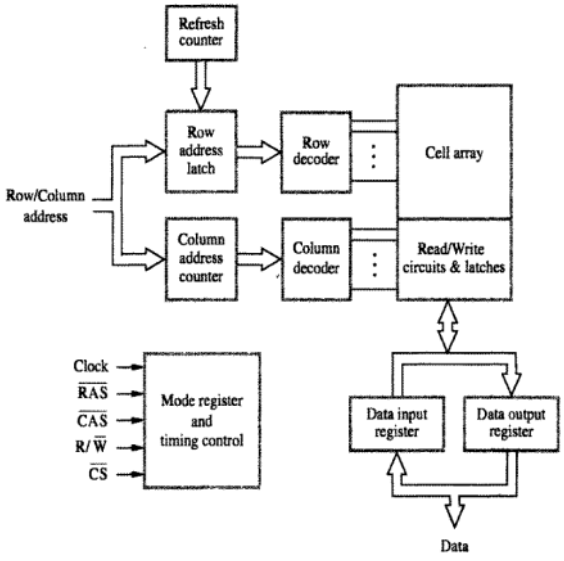


Figure 5.8 Synchronous DRAM.

In this type DRAM the control signal is internally generated by the RAM
This is typically useful in burst mode or page transfers where the whole page operation is handled by the RAM alone without processor intervention.

Most synchronous details are omitted and may be read from carl hamcher

Structure of larger memories

09 February 2021 03:29 PM

This is how we make a memory of 2M words each word is 32 bits.

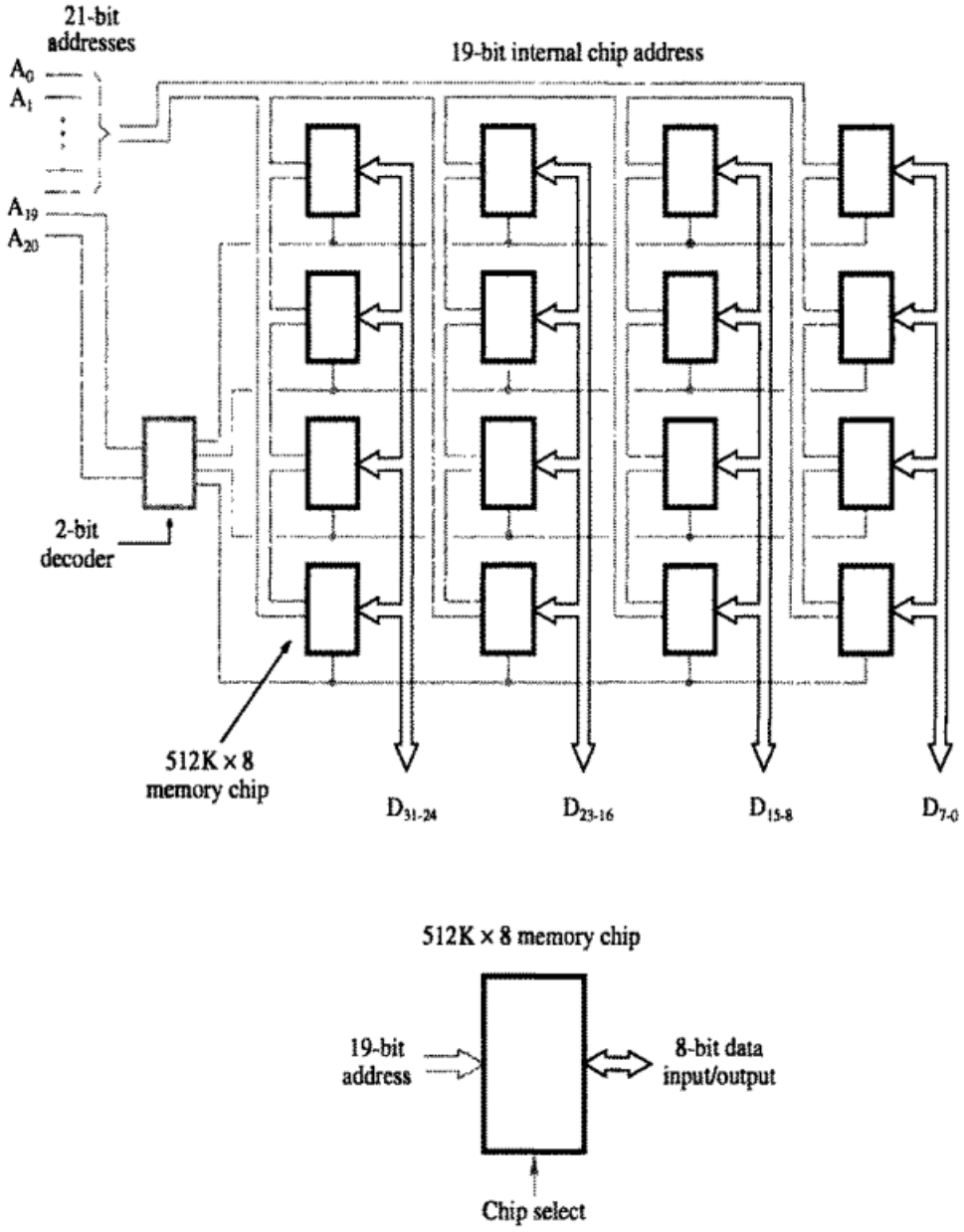


Figure 5.10 Organization of a 2M × 32 memory module using 512K × 8 static memory chips.

Each 512x8 chip is connected to a select lines
The complete address to a word is 21 bit long
Where the 2 bit decoder selects a single line which activates each chip on line
Each of the individual chip requires a 19 bit address to access a byte of data
Since the 4 segments of a word is equally distributed and are stored in different chips on same line
So all the 4 bytes are strobed hence giving a word of data.

How the memory is integrated with mother boards using SIMM and DIMM is omitted and given carl hamcher

Memory system consideration

09 February 2021 03:52 PM

SRAMs are costly and generally used in caches
DRAMs are less costly and generally used for implementing main memories

Memory controller

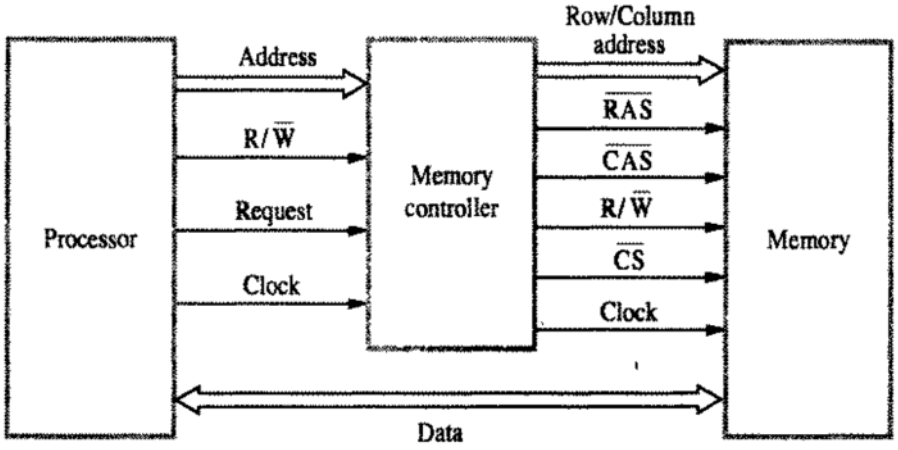


Figure 5.11 Use of a memory controller.

To reduce the number of pins on a main memory it uses a multiplexed address input
The row address is sent first and are latched followed by the column address.

A processor typically sends the whole address at once a memory controller sends the row and column address to the main memory and also all the required control signals.

In certain DRAMs which do not have self refreshing capability the controller is also responsible to refresh the memories.

Refresh Overhead

All dynamic memories have to be refreshed. In older DRAMs, a typical period for refreshing all rows was 16 ms. In typical SDRAMs, a typical period is 64 ms.

Consider an SDRAM whose cells are arranged in 8K (=8192) rows. Suppose that it takes four clock cycles to access (read) each row. Then, it takes $8192 \times 4 = 32,768$ cycles to refresh all rows. At a clock rate of 133 MHz, the time needed to refresh all rows is $32,768/(133 \times 10^6) = 246 \times 10^{-6}$ seconds. Thus, the refreshing process occupies 0.246 ms in each 64-ms time interval. Therefore, the refresh overhead is $0.246/64 = 0.0038$, which is less than 0.4 percent of the total time available for accessing the memory.

Rambus memory is omitted and in carl hamcher

ROMs

09 February 2021 04:51 PM

Their are certain programmes like the boot programme which is responsible for loading the OS to the memory which are required to be stored into the memory for indefinite period of time even when power goes off for that we use ROM

They are of following types
ROM
PROM
EPROM
EEPROM
Flash memory

Speed Size Cost

09 February 2021 05:02 PM

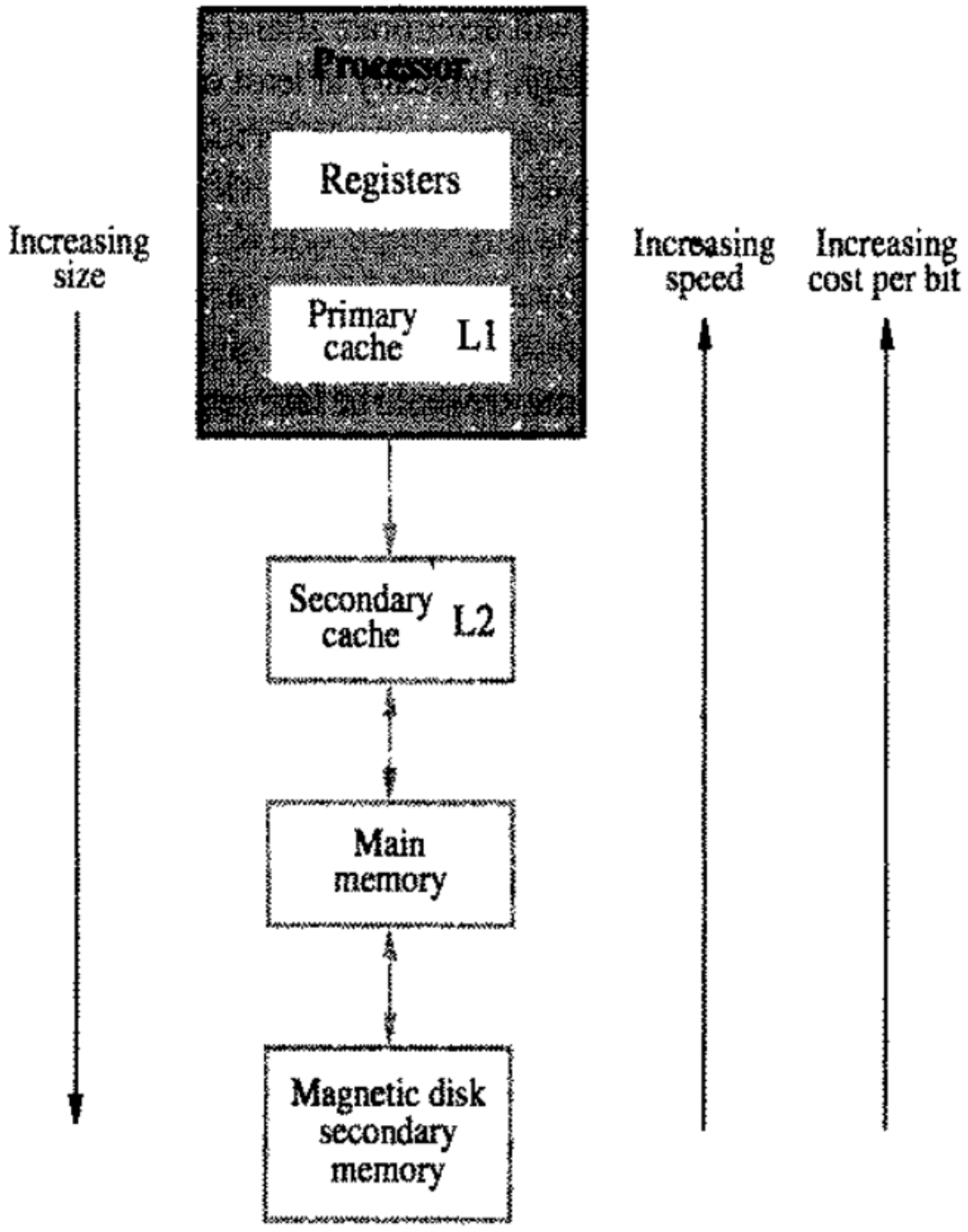


Figure 5.13 Memory hierarchy.

Cache memories

09 February 2021 05:14 PM

- Speed of main memories is very slow as compared to the processors.
- To increase the efficiency a much faster cache memory is introduced between processor and main memory.
- This essentially makes the main memory appear faster to the processor then it really is.

- The effectiveness of cache is due to the property of computer programmes called 'locality of reference'.
    - This property manifests itself in two ways:
        - Temporal : same programme being accessed frequently. (loops, recursion etc.)
        - Spatial : the programme to be executed next resides close in memory.

Basic working of cache

- Programme issues a read request to the memory.
- If it resides in the cache it is fetched other wise a cache block/ cache line is transferred from main memory to cache where an existing page is replaced with new one.

- The processor does not knows explicitly about the existence of cache it simply issues a read write request it is the cache control circuitry which must determine whether the requested info is in cache.
- If the info resides in cache a read write occurs and read write hit is said to have taken place.
- When an addressed word does not resides in cache a read miss is said to have occurred.

Some concepts

Mapping functions: the correspondence between main memory blocks and cache blocks are specified by the mapping function.

Replacement algorithms: the collection of rules which decides what block to be replaced from cache when there is no space for new one is called replacement algorithms.

When the processor issues a write request this can be handled in to ways

Write-through protocol: cache and main memory updated simultaneously.
Write back/copy back protocol: cache is updated and when the page is about to be replaced from cache it is written back to main memory and then replaced. A bit called dirty bit/modified bit is responsible for marking the pages that are to be written back. (if the word to be updated is not in cache it must be brought to the cache and updated in the cache.)

When a read miss occurs the block containing the word is transferred form main memory to cache the word requested by the processor may be sent as soon as it is read from the main memory - load through/early restart.
Or when it comes to cache then it may be sent from cache.

- Control bits in cache block
    - Valid bit: indicates that a block is valid.
        - if main memory is updated and their is no need for storing it in cache then the outdated block in cache will be not valid.
    - Dirty bit: if the cache block is updated it is set to 1 to indicate that main memory needs to be updated before its removal.

- POINTS
    - DMA transfers bypasses cache

Mapping functions

10 February 2021 09:01 AM

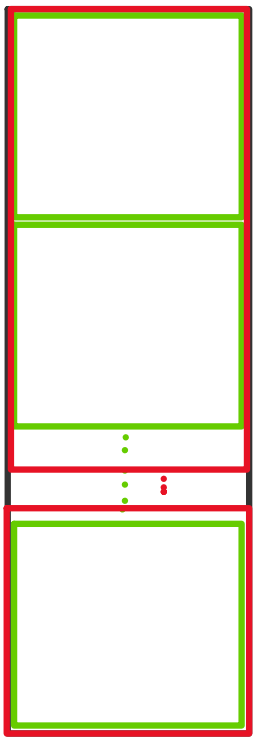
To discuss various functions in detail we consider a small example here:

- A primary memory of 64K words = pow(2,16) words
  - Hence it uses a 16 bit address
  - We divide them equally into pow(2,12) blocks = 4096 blocks
  - Each block of 16 words
- A cache of pow(2,11) words
  - Which means 128 blocks = pow(2,7) blocks
  - Each block of 16 words

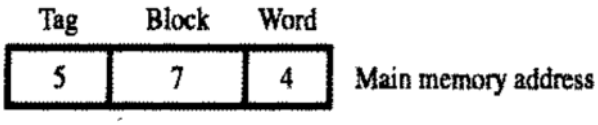
Direct mapping

10 February 2021 01:25 PM

- Block j of main memory is mapped to j%128 block of cache (cache has 128 blocks in total)
  - So 0, 128, 256 of main memory.... Goes to block 0 of cache
  - 1, 129, 257 of main memory.... Goes to block 1 of cache and so on.



- Address correspondence in main memory
  - The main memory uses a 16 bit address
  - Let us divide the memory blocks of main memory like the figure on left
    - Green -> block (containing 16 words)
    - Red -> group of 128 blocks (same as cache capacity)
    - No of green box per red box is 128 = 128 pow(2,7)
    - No of red box = (total no of green box / 128) = (pow(2,12)/pow(2,7))=pow(2,5)
    - The highest order 5 bits of the RAM address corresponds to the pow(2,5) red boxes
    - Next 7 bits corresponds to pow(2,7) blocks inside a red box
    - Next 4 bits corresponds to pow(2,4) word inside a green box



- Address correspondence in cache
  - Cache is much like a main memory
  - In our example it has pow(2,11) words
  - This means 11 bit address is required to address cache
  - At a time only 128 blocks can be stored in cache
  - Every block is uniquely mapped to cache by j%128
  - The mod operation slices down the tag part of 16 bit RAM address, we are left with 11 bit address with which cache is addressed
  - The address with same block + word part and different tag part would mean the came to cache
  - To avoid confusion with data the tag is also stored in cache along with data
  - So when an address is looked up in cache it is looked up in the actual 11 bit address and the tag is matched if it does not matches we bring the data from ram

- Replacement algorithm
  - Simple, check in cache if data is not there replace ut with the required data.

Associative mapping

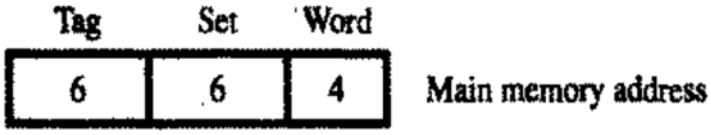
10 February 2021 03:00 PM

- A block of the main memory can be placed in any cache block.
- A 12 bit tag is also needed to be stored along with the data.
- This 12 bit tag is to identify pow(2,12) blocks on main memory.
- So the tag is searched to see if block is in the cache if it is then the last 4 bit identifies the word required.
- If the block is not their a replacement algorithm decides which block is to be replaced.

Set associative mapping

10 February 2021 03:08 PM

- This is a combination of direct and associative mapping
- We give an example of 2 way associative mapping
    - We group two blocks in cache into a set
    - There will be pow(2,7)/2 = pow(2,6) such sets



    - The set will be identified by 6 bits
    - Their will have to be a 6 bit tag associated with the data
    - This identifies a block.

Replacement algorithms

10 February 2021 03:27 PM

There are mainly three replacement algorithms

- Randomized
- FIFO - the first block to enter will first replaced
- LRU (least recently used) - the block that is least recently used will be replaced
    - Implementation
      Each block has a counter associated
      When hit occurs the counter of block which is used is reset and all others is incremented
      When a miss occurs the block with highest counter value is replaced the new block's counter is reset
      All other's are incremented.

Performance Considerations

10 February 2021 04:33 PM

The whole story of cache is a result of lower the cost/performance ratio.

This section discusses more on performance

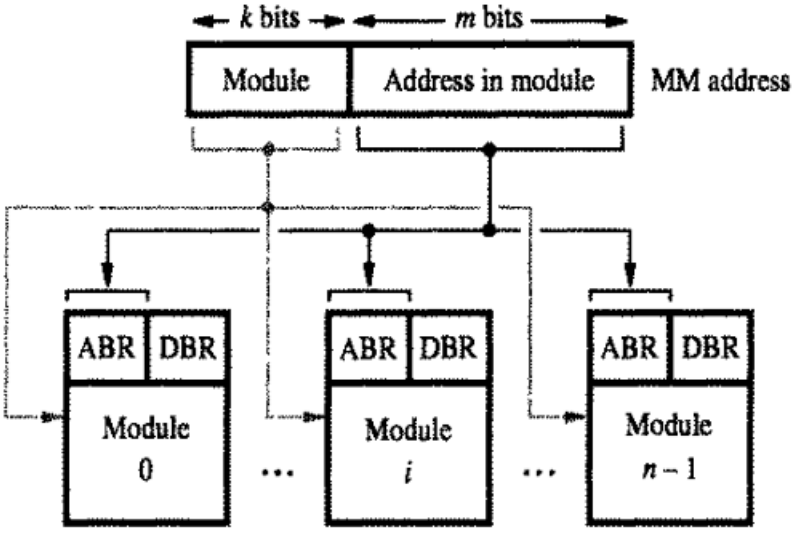
Interleaving

10 February 2021 04:35 PM

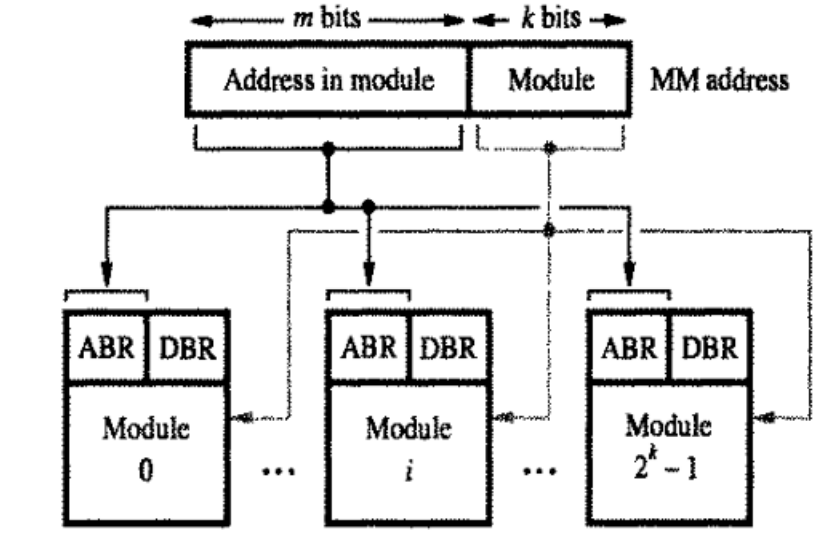
In computer systems whenever a faster component interacts with a slower component the speed is limited by the slower one, to overcome this we may design the slower system to work parallelly.

In case of memory we may use multiple modules of small chips with separate address buffer and data buffer registers.

Here given are two such systems.



(a) Consecutive words in a module



(b) Consecutive words in consecutive modules

Figure 5.25 Addressing multiple-module memory systems.

Due to locality of reference (operating on contagious memory locations) the first approach is

not as effective because when consecutive words are to be fetched (which generally happens) and consecutive words are in same module it will not increase speed.

In the second approach however consecutive words are stored in separate modules so it greatly increases speed.
In this case however a total of pow(2,k) module is required if k bits address the module if not some words will be absent in between.

Carl hamcher explains more taking an example and counting clock cycles

Hit rate and miss penalty

10 February 2021 05:01 PM

Hit rate : total no of hit/total no. Of attempts
Miss rate : total no of miss/total no. Of attempts
Miss penalty: the total time required to bring a data from lower hierarchy memory unit to higher hierarchy unit in case of a miss.

Problems on it in carl hamcher

Omissions

10 February 2021 03:45 PM

Section 5.5.3 of carl hamcher gives an example of programme and explains in detail the contents of cache in three kinds of mapping is omitted here

Section 5.5.4 examples of caches in commercial processors

Sections 5.6.3 / 5.6.4

Left till now

- Carl hamcher 5.7-5.10

Addition and subtraction

13 February 2021 06:28 PM

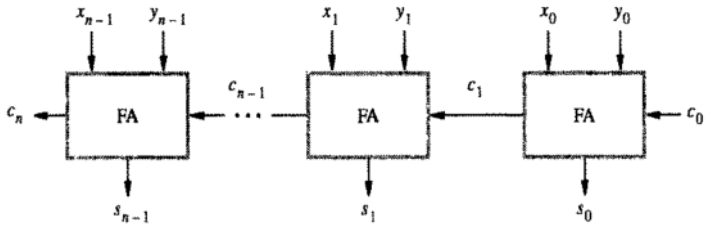
- Two numbers are added as two binary vectors
  - Each step requires the two bits and a carry from previous step

x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \bar{x}_i\bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i\bar{y}_i\bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

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

  - This circuit is implemented by a full adder.
  - A cascaded FAs makes a n-bit-ripple-carry-adder



(b) An *n*-bit ripple-carry adder

    This circuit is responsible for binary vector addition
  - A subtraction operation occurs as addition of one normal bit vector (positive no.) and one twos compliment bit vector (negative no.)
  - Overflow is checked by checking the result of $c_n \oplus c_{n-1}$ if it is 1 overflow occured.

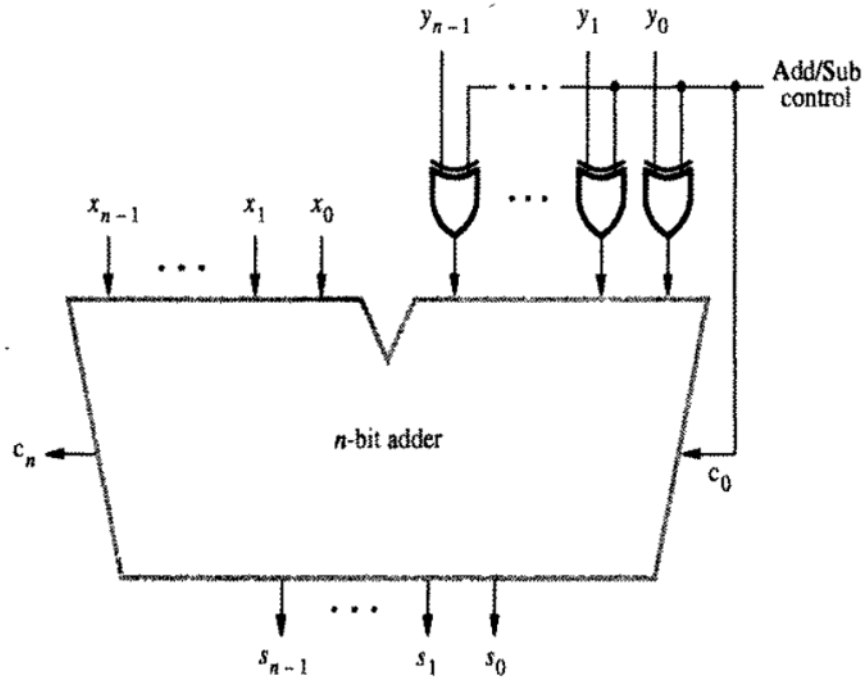


Figure 6.3 Binary addition-subtraction logic network.

This the complete circuit of a n-bit adder

- The add/sub control line is set to 0 if the number is to be added .
- If the no is to be subtracted the add/sub line is set to 1 making the y vector as 1s complement by the XOR gates and giving an initial carry of 1 which ultimately makes y bit vector 2 complemented.

Disadvantages

- The n-bit ripple-carry adder is very slow

Carry look ahead addition

13 February 2021 07:31 PM

- The n-bit ripple carry adder is slow because of the equation

  $C_{i+1}=x_iy_i+x_ic_i+y_ic_i$

  Here the x_i and y_i is available at that instant but c_i is required to be calculated therefore needs a lot of time since it ultimately depends upon the carry of the initial input.

- Speeding up the carry

  We write the carry equation as follows

  $C_{i+1}=x_iy_i+(x_i \oplus y_i)ci$

  This also gives the same result

  X_iy_i -> G_i (generator)

  $(x_i \oplus yi)$-> P_i (propagator)

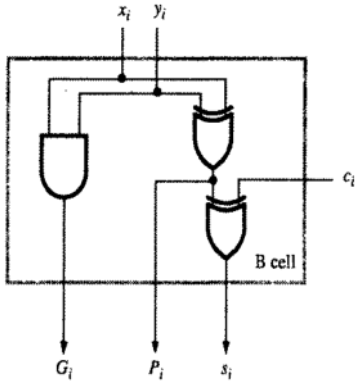
  Expanding the carry equation so it does not depend upon any thing other than x_i ,y_i ,c_0 we get the following

  $$c_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \cdots + P_iP_{i-1}\cdots P_1G_0 + P_iP_{i-1}\cdots P_0c_0$$

  The whole idea is to directly give carry to individual adders so that a particular adder does not wait for the adder before that.

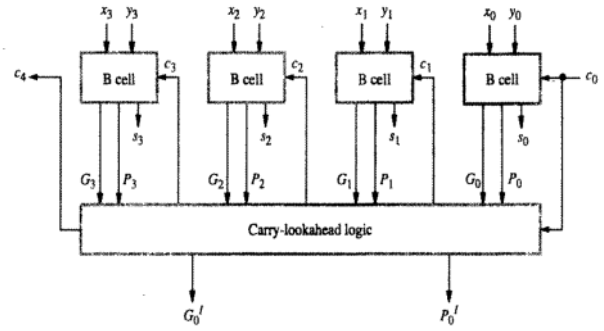
Implementation

The individual G_i and P_i is calculated by bit stage cell



(a) Bit-stage cell

Design of a 4 bit adder



(b) 4-bit adder

Figure 6.4 4-bit carry-lookahead adder.

The carry lookahead logic just contains the following equations

$$c_1 = G_0 + P_0 c_0$$
$$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$
$$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$
$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

The disadvantages of this adder is that we can not extend it directly above 4 bits because of gate fan in constraints.

Higher level generate and propagate functions

To extend this circuit further for more no. Of bits we may use the ripple approach but still lookahead approach is much faste r.

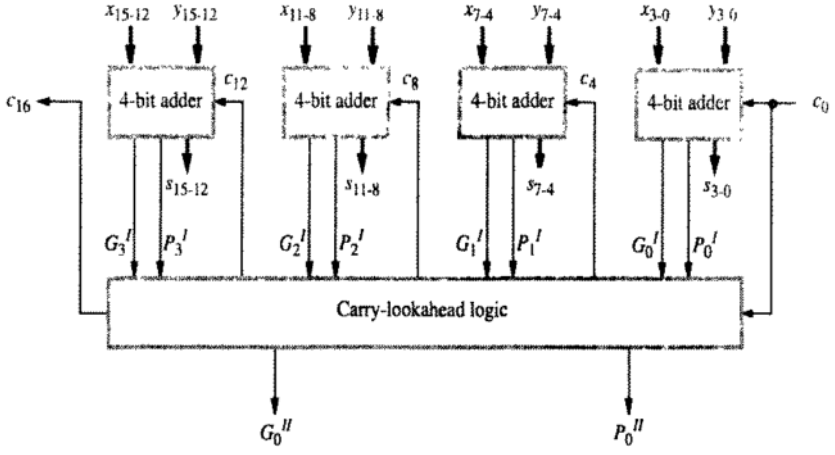


Figure 6.5 16-bit carry-lookahead adder built from 4-bit adders (see Figure 6.4*b*).

This uses 4 four bit lookahead logic to combine them and with a higher level look ahead logic.

Consider

$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

The term containing C_0 is $P_3P_2P_1P_0$ is called the higher level propagate function P_0^I
The rest of 4 terms is called generate function G_0^I

Similarly

$$c_{16} = G_3^I + P_3^I G_2^I + P_3^I P_2^I G_1^I + P_3^I P_2^I P_1^I G_0^I + P_3^I P_2^I P_1^I P_0^I c_0$$

Gate delays are omitted an in carl hamsher

Multiplication

15 February 2021 10:51 AM

Multiplication of positive numbers

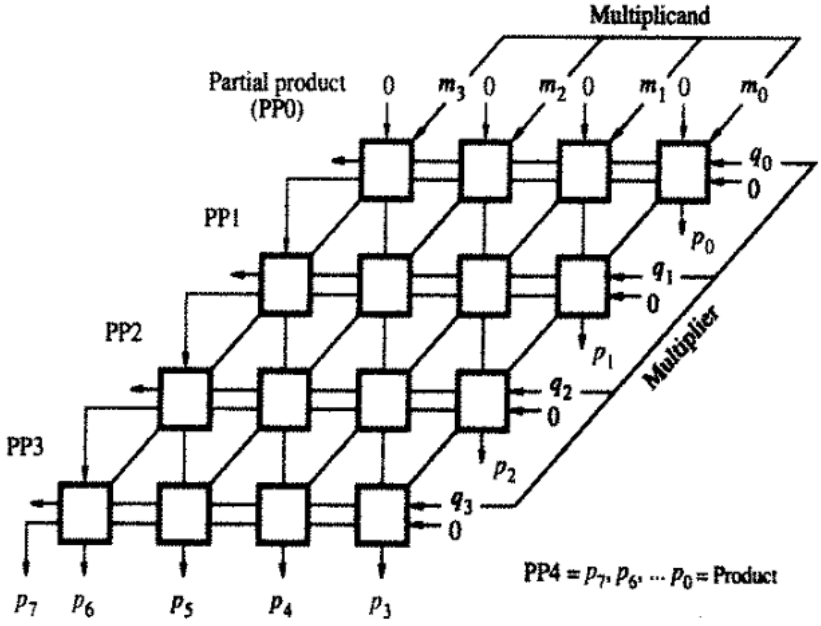
15 February 2021 10:35 AM

Binary multiplication is performed in the same way as usual multiplication

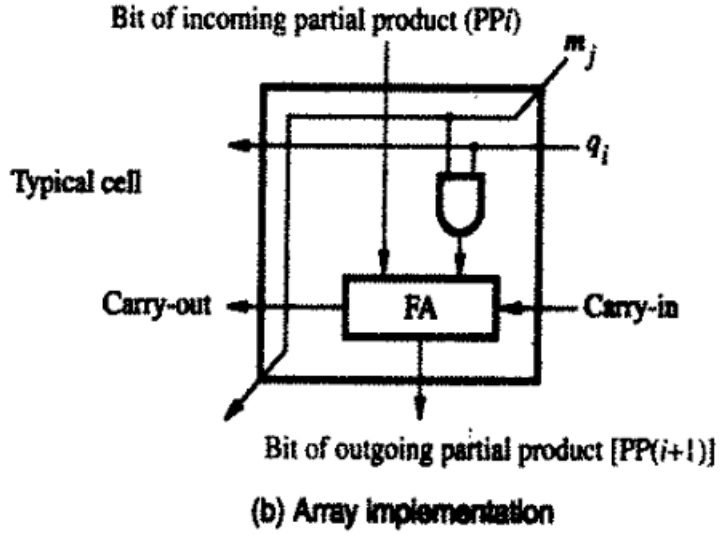
```
          1 1 0 1      (13) Multiplicand M
        × 1 0 1 1      (11) Multiplier Q
        ---------
          1 1 0 1
        1 1 0 1
      0 0 0 0
    1 1 0 1
    ---------------
1 0 0 0 1 1 1 1      (143) Product P
```

(a) Manual multiplication algorithm

This manual method is replicated by using a n x n cells as shown



Each cell is the following



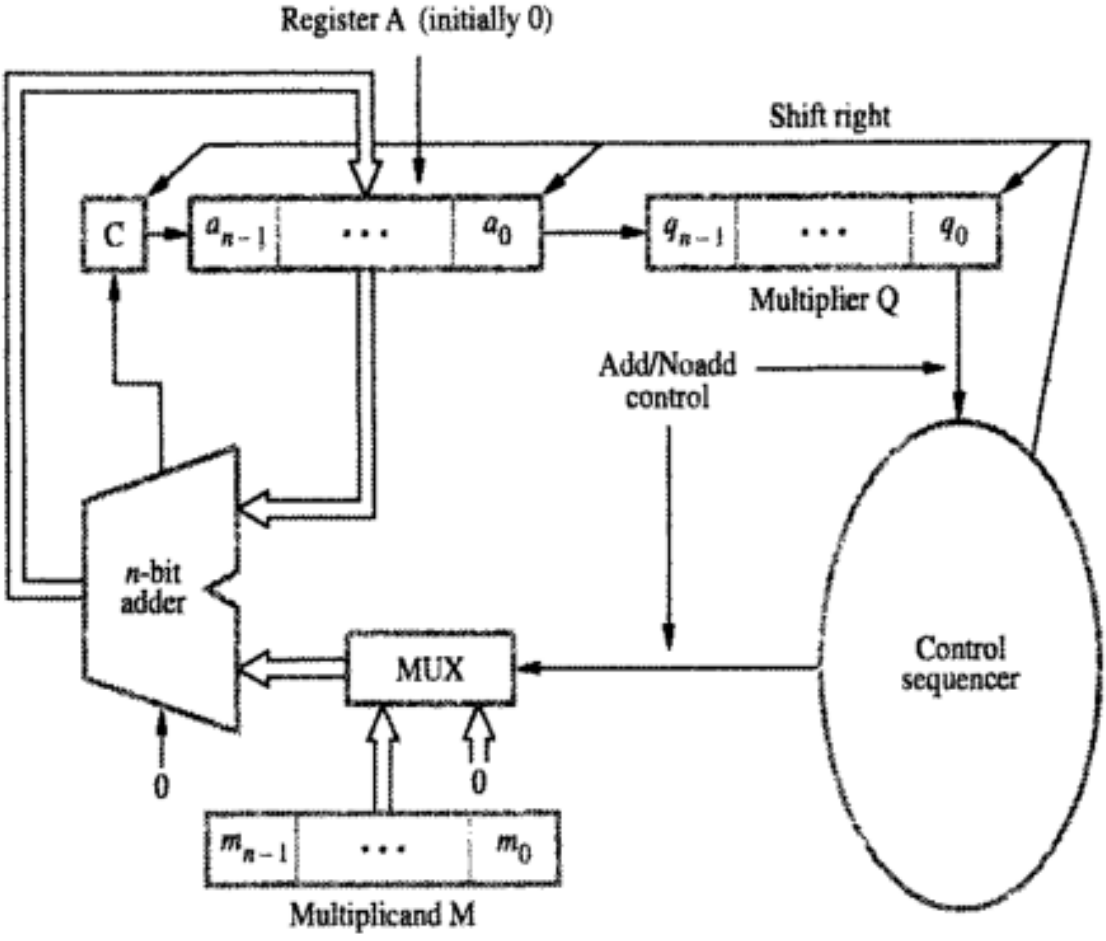
(b) Array implementation

(these figures are self explanatory)

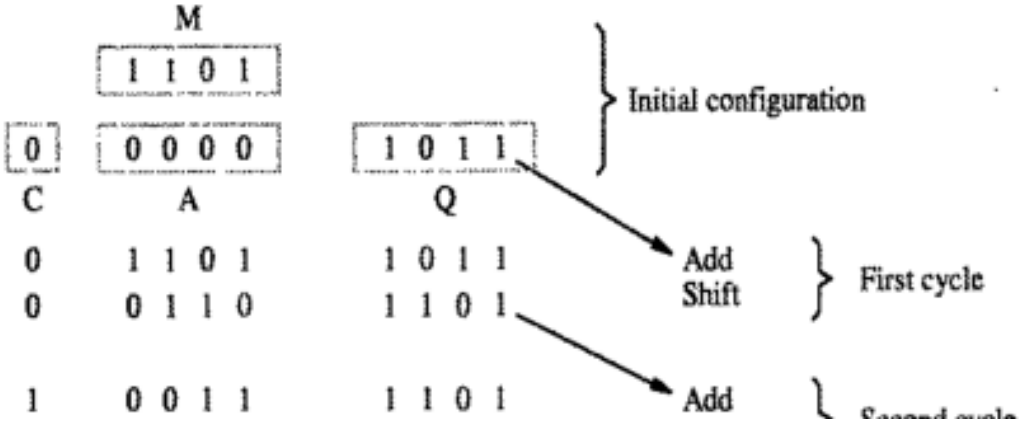
Improvements on previous methods

1. Store multiplicand in register
2. Store multiplier in a register
3. Initialise the result register to 0
4. For each LSB of multiplier
    a. Add the multiplicand to result register contents if LSB is high
    b. Left shift the multiplier

This implemented by the following



(a) Register configuration



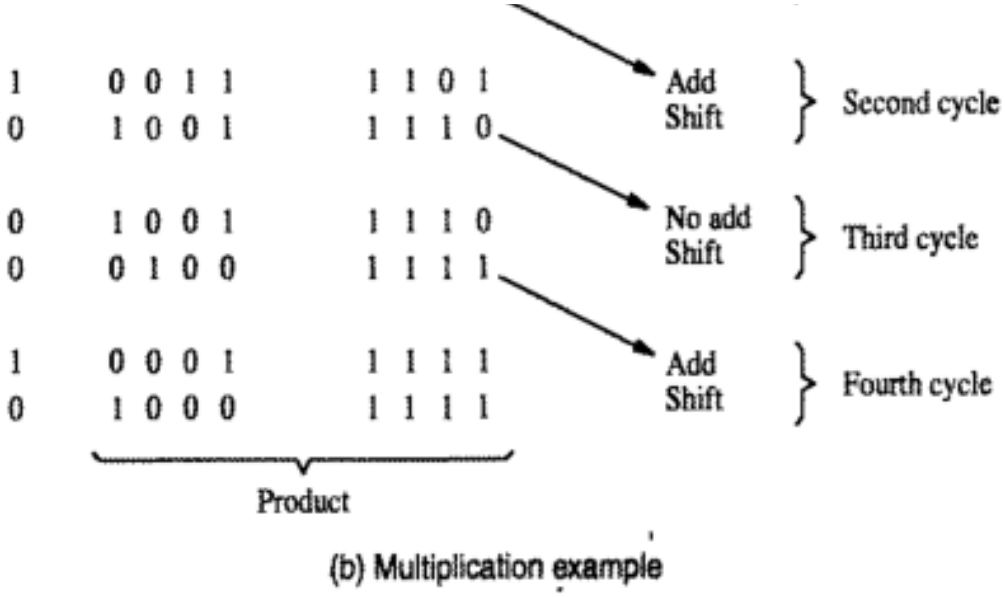


Figure 6.7 Sequential circuit binary multiplier.

Since after each addition the result will increase one bit in size we use the multiplier register too to hold half bits of the result.

Signed number multiplication

15 February 2021 10:51 AM

- The following facts can be easily proved
  - A(normal) * b(normal) = C(normal)
  - A(normal) * b(2 complemented) = C(2 complemented) ..(signed bit is extended to twice the bit length)
  - A number 2 complemented twice is back that number

Multiplication of signed no is performed as follows

Single signed no.

Normal multiplication but with signed bit extension

Both signed no.

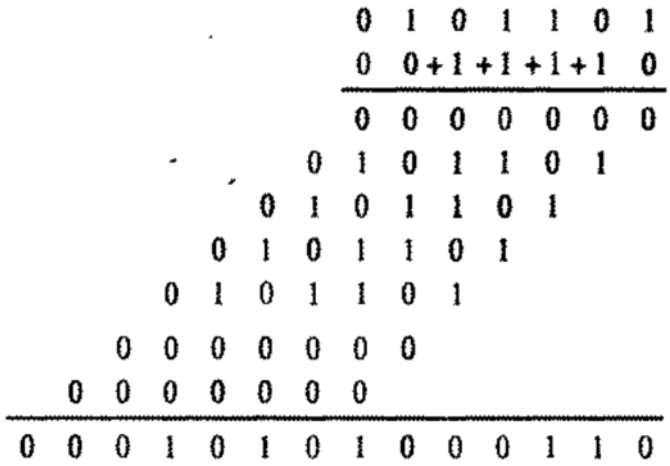
Both the number is 2 complemented again effectively un complementing them then is normally multiplied

Booth algorithm

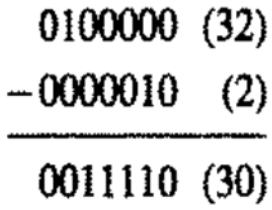
15 February 2021 11:22 AM

1. Partial product/result is initialized to 0
2. The multiplier is scanned from LSB to MSB
3. if a toggle occurs from 0 to 1 the result is added with the 2 complement of the appropriately shifted multiplicand (effectively subtracting it from the result)
4. Else if a toggle occurs from 1 to 0 the result is added with appropriately shifted multiplicand (effectively adding it to the result)
5. Else if no toggle occurs do nothing.

A comparison between normal and booth algorithms
Normal method:

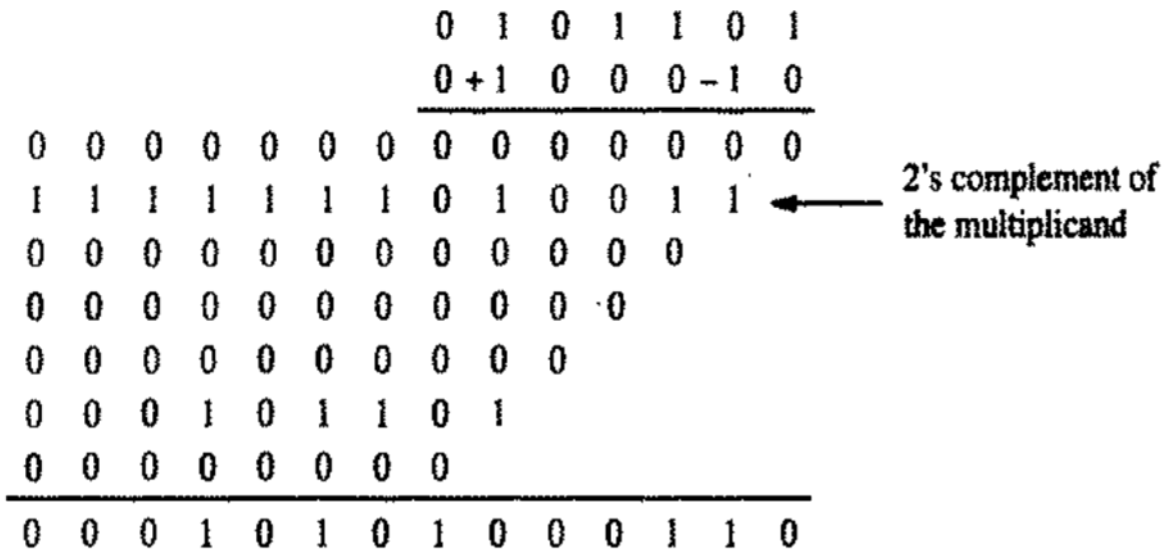


Now the multiplier can be written as



30 = 32 -2
Hence reducing the no. Of ones and also operations
Thus
30*45 = (32-2)*45
The converting of a normal multiplier to booth multiplier as shown below is just a clever approach to implement the above idea.

Normal multiplier 0 0 1 1 1 1 0
Booth multiplier 0 +1 0 0 0 -1 0
Booth method:



Conversion of normal multiplier to booth multiplier

Scan the multiplier from LSB to MSB
If LSB is 1 then assume the toggle has occured from 0 to 1
If a toggle occurs from 0 to 1 write a -1
If a toggle occurs from 1 to 0 write a +1
Else if no toggle occurs write a 0

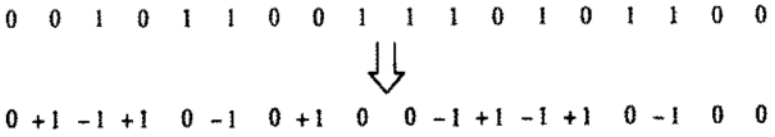


Figure 6.10 Booth recoding of a multiplier.

The more the number of 0 the more good booth algorithm will run
On an average it performs the same as normal algorithm

The correctness of booth algorithm for negetive no is omitted and in carl hamsher

Fast multiplication

15 February 2021 04:49 PM

Bit-pair recoding of multipliers

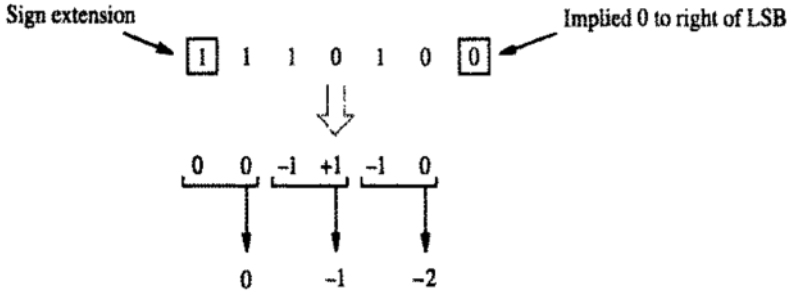
- In this technique two bit of the Booth multiplier is reduced to equivalent one (the result of using the pair is same as using the one bit)

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position i
$i+1$	i	$i-1$	
0	0	0	$0 \times M$
0	0	1	$+1 \times M$
0	1	0	$+1 \times M$
0	1	1	$+2 \times M$
1	0	0	$-2 \times M$
1	0	1	$-1 \times M$
1	1	0	$-1 \times M$
1	1	1	$0 \times M$

(b) Table of multiplicand selection decisions

This table may be verified
LSB is 0th position and MSB is highest

An example :



(a) Example of bit-pair recoding derived from Booth recoding

```
      0  1  1  0  1   (+13)
    × 1  1  0  1  0   (–6)
    ---------------
```

```
                  0  1  1  0  1
                  0 –1 +1 –1  0
               ---------------
   0  0  0  0  0  0  0  0  0  0
   1  1  1  1  1  0  0  1  1
   0  0  0  0  1  1  0  1
   1  1  1  0  0  1  1
   0  0  0  0  0  0
---------------------------------
   1  1  1  0  1  1  0  0  1  0   (–78)
```

```
                  0  1  1  0  1
                  0  –1    –2
               ---------------
   1  1  1  1  1  0  0  1  1  0
   1  1  1  1  0  0  1  1
   0  0  0  0  0  0
---------------------------------
   1  1  1  0  1  1  0  0  1  0
```

Figure 6.15 Multiplication requiring only $n/2$ summands.

Carry-save addition of summands

15 February 2021 04:56 PM

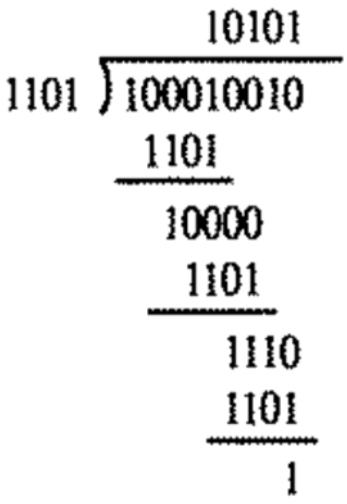
Omitted and in carl hamsher

Integer Division

15 February 2021 06:49 PM

Manual method for binary division
Until dividend bits are left

1. The divisor is positioned appropriately with respect to dividend and a subtraction is performed
2. If the result of subtraction is negetive
    a. we put a 0 in quotient and restore the placing of divisor
    b. We then take another bit of dividend and go to step 1
3. Else
    a. We put a 1 in quotient
    b. We then take another bit of dividend and go to step 1



Restoring division

16 February 2021 07:06 AM

- Restoring division is a convenient way of doing binary division in a machine
- Steps:
- Assume A is a register
  Q is a register holding n-bit positive dividend initially
  M is a register holding n-bit posetive divisor

1. Repeat n times
   a. Shift A and Q left one binary position
   b. A:=A-M
   c. If A is negetive
      i. Set LSB of Q to 0
      ii. A:=A+M(restore A)
   d. Else
      i. Set LSB of Q to 1

At the end quotient is in Q
Remainder is in A

The following circuit does the job

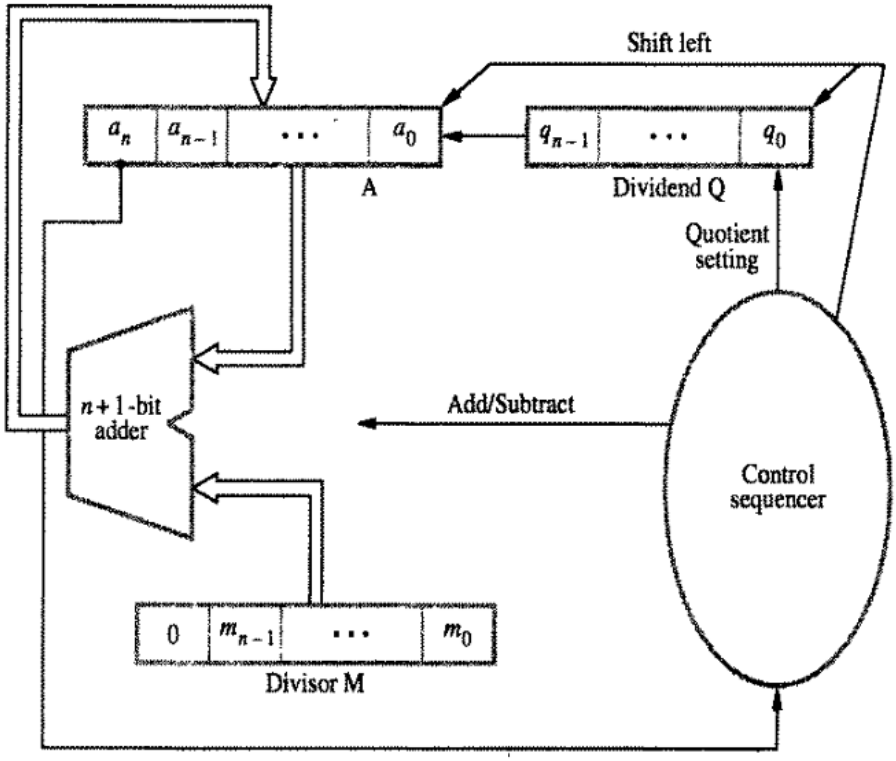


Figure 6.21 Circuit arrangement for binary division.

```
      10
11 ) 1000
     11
     --
     10
```

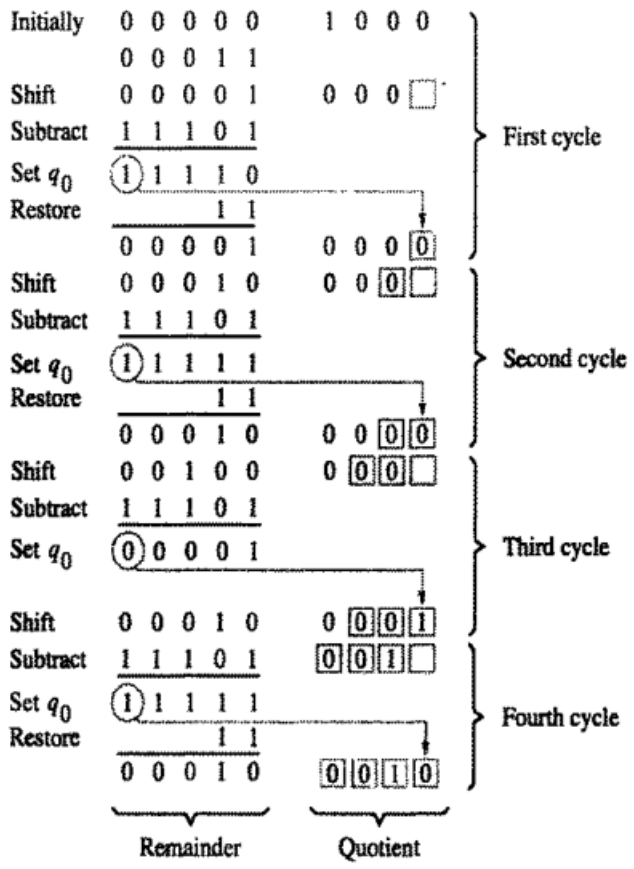


Figure 6.22 A restoring-division example.

Non-restoring division

16 February 2021 07:17 AM

What we actually do in restoring division

- If A is posetive we shift left and subtract M
  - i.e. A:=2A-M
- If A is negetive we add M (restore) then shift left and subtract M
  - i.e. A:= 2(A+M)-M=2A+M

This is what we implement in non restoring division

1. Repeat n times
   a. If A is posetive
      i. Shift A and Q left
      ii. A:=A-M
   b. Else
      i. Shift A and Q left
      ii. A:=A+M
   c. If A is posetive
      i. LSB of Q =1
   d. Else
      i. LSB of Q =0
2. If A is negative
   a. A:=A+M (to leave a posetive remainder)

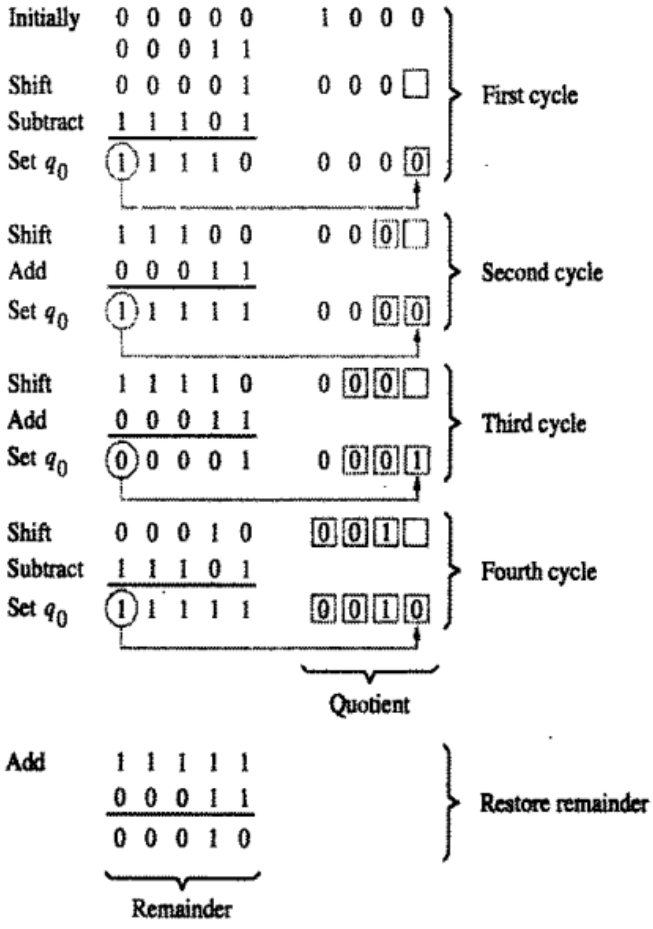


Figure 6.23 A nonrestoring-division example.

Floating point numbers

16 February 2021 07:34 AM

- For integer representation in computers we assume a binary point (~~decimal point~~) after its LSB.
- This binary point in integers remain fixed and so integers are called fixed point numbers
- For a general number where the binary point can be any where, the binary point is adjusted automatically as the calculation proceeds.
- The binary point is set to float in such numbers and is thus called floating point numbers.
- Usually a decimal number in scientific notation is written as follows
    - 6.0247×10^{23}
    - $A \times 10^{c}$
    - A may be negative or posetive (mantissa) usually written with 5 significant digits and when the decimal point is written just after first digit the number is said to be normalized.
    - C may be posetive or negative (exponent)

IEEE standards

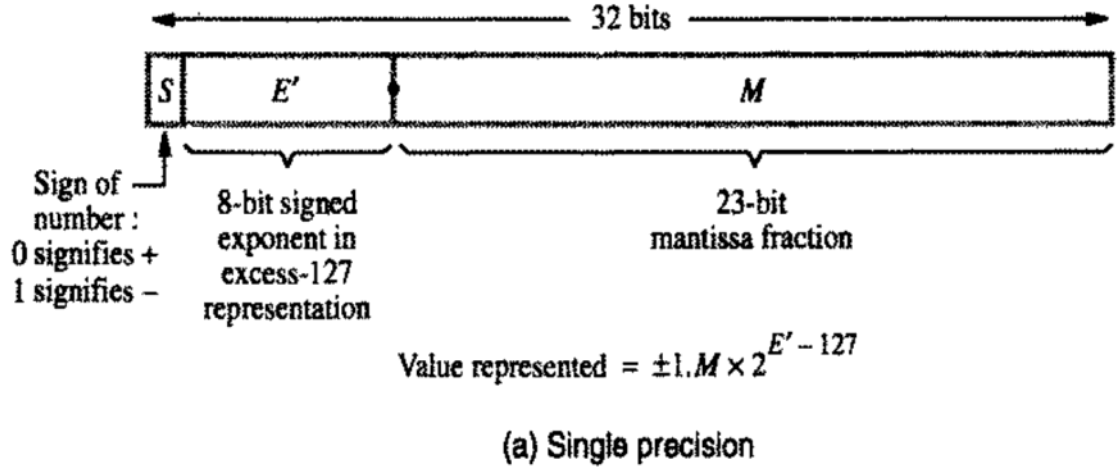
16 February 2021 07:55 AM

Some points

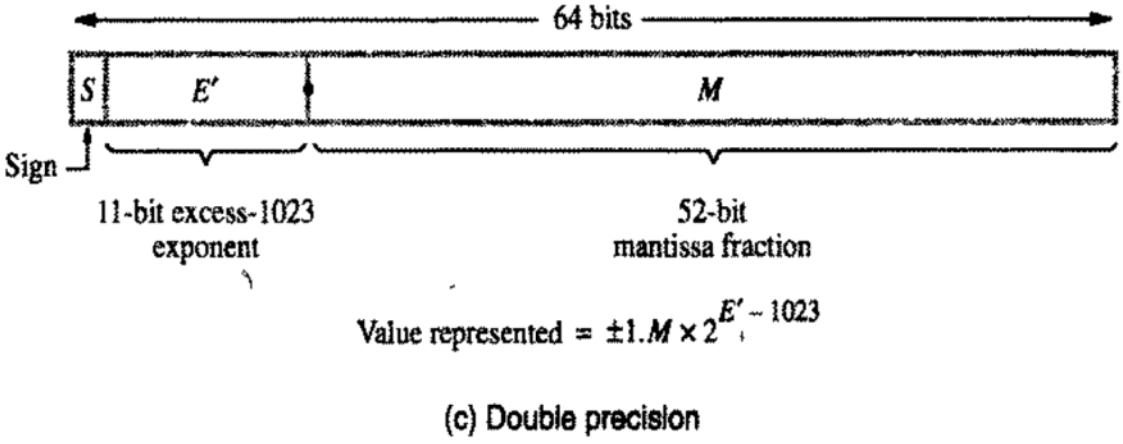
- In the binary notation the exponent implied is 2 and not 10. (makes shifting decimal points easy just shift and adjust the exponent)
- The number is represented in normalized form
- The first bit of binary mantissa in a normalized form is always 1 hence it is omitted.
- To avoid dealing with sign in the exponent part instead of exponent E, E+127 is stored in case of single precision and E+1023 is stored in case of double precision.

This slandered provides for two different precession (range)

1. Single point
2. Double point

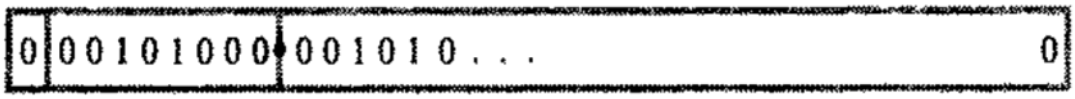


(a) Single precision



(c) Double precision

An example



$$\text{Value represented} = 1.001010\ldots0 \times 2^{-87}$$

(b) Example of a single-precision number

Special values

- Exact 0 : +/- zero can be represented
- Infinity : result of dividing by zero +/- infinity can be represented
- Denormal numbers : no. Less than normal numbers
    - The 1 at the MSB of mantissa is not implied.
    - It is done for gradual under flow.
- Nan : i or 0/0

E(exponent)	M (mantissa)	value
0	0	Exact 0
255(max)	0	infinity
0	Not 0	Denormal numbers
255(max)	Not 0	NaN

Exceptions

- Over flow : exponent of a number less than range
- Under flow : exponent of a number more than range
- Divide by zero :
- Inexact : number requires rounding in order to be stored in normal formats
- Invalid : 0/0 or root(-1) has occured

Omissions

16 February 2021 09:01 AM

6.7.2, 6.7.3, 6.7.4 carl hamsher

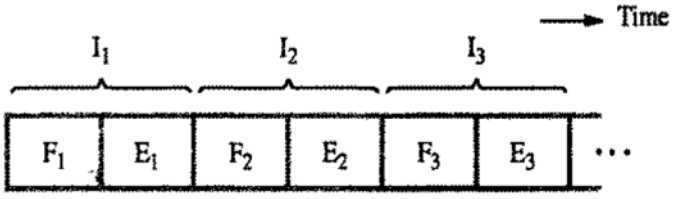
18 March 2021 06:25 PM

Horizontal va vertical microprogrammed control

Basix

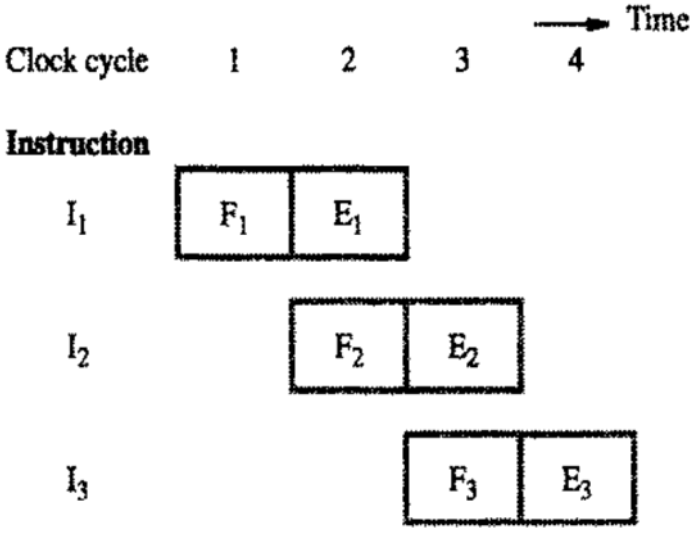
17 February 2021 03:28 PM

- A particular instruction needs yet some miniature instructions to execute it.
- All machine instruction is carried out by the cycle of
  - Fetch
  - Decode
  - Read
  - Execute
  - Write
- For simplicity we consider here only two sub steps
  - Fetch
  - Execute
- Normally (not pipelined) the execution will take place as follows



(a) Sequential execution

- After the fetch the data fetched is stored in an inter-stage buffer.
  - Like MDR etc.
- A pipelined execution looks like the following



(c) Pipelined execution

This is a two stage execution cycle

The clock cycle required is T= no. of stages + no. of instruction -1

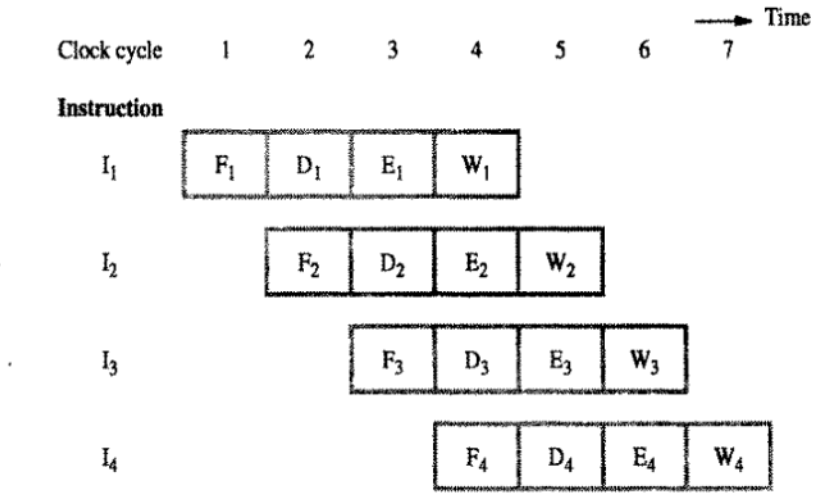
= 2+3-1 = 4

For sequential execution(not pipelined) T= no. of instruction*2

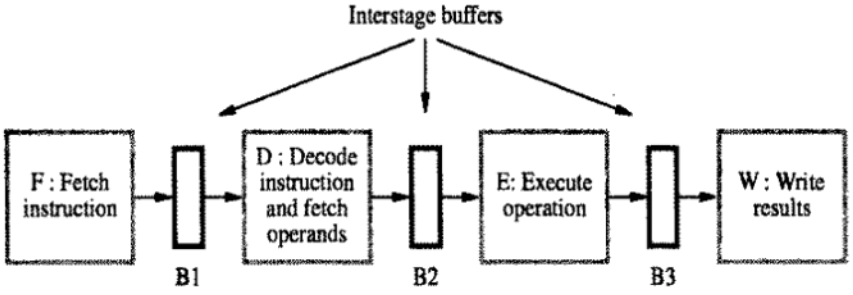
- To give summary
  - It is like two separate units inside the processor are working separately
  - In three instruction process each instruction can be accomplished by two steps
  - The fetch part is continuously fetching the data and passing it to execute part.
  - Now the execute part starts doing its job meanwhile the fetch part is busy fetching other

data.

- For more no of sub steps it will be the following scenario



(a) Instruction execution divided into four steps



(b) Hardware organization

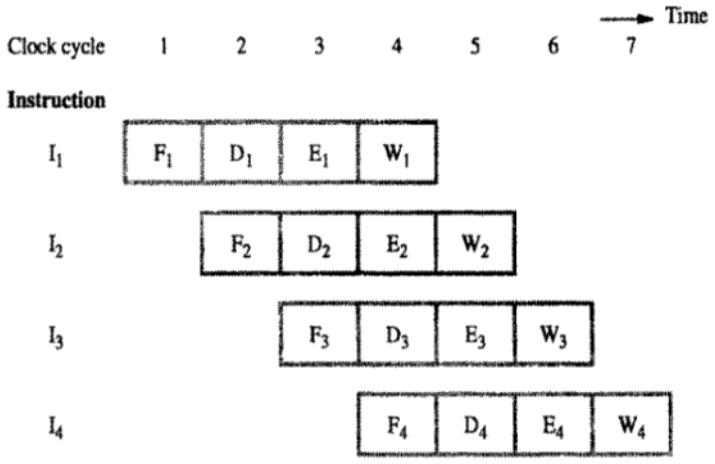
Figure 8.2 A 4-stage pipeline.

- These steps are effective only if every process takes almost equal amount of clock cycles.
  - The clock cycles required to complete each stage must be the maximum of all the stages in terms of time
    - So if fetch requires 10 clk. Cycles (absence of cache or cache miss).
    - All the other steps will have to be tune for 10 clk. Cycles which is not much better than non pipelined execution.

Performance

19 February 2021 09:48 AM

- If every thing goes well the following is the scenario



(a) Instruction execution divided into four steps

- But this is not ideal very often a particular stage in a particular instruction takes a longer time and this is the scenario.

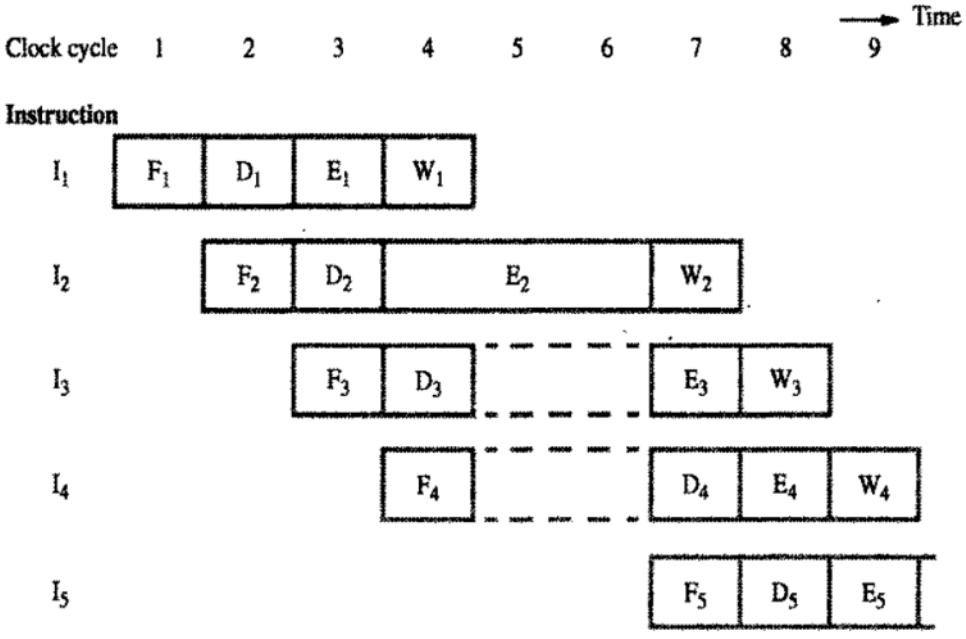


Figure 8.3 Effect of an execution operation taking more than one clock cycle.

  - Due to execution delay in third steps all the steps below it will be delayed.
  - Some stages below will be stalled (do nothing) for some time. (here in 5 clk. Cycle decode and fetch are stalled 4th and 5th instruction).
  - The pipelined execution stalled for two clk. Cycles and normalcy resumes in 7th clock cycle.
- Any condition that causes the pipe line to stall is called a hazard.
  - Data hazard (delay due to data unavailability)
  - Instruction hazard (delay due to instruction unavailability)
  - Structural hazard (two stages requiring same resource (memory for example))
    - To reduce structural hazards separate instruction and data cache are used.

Data hazards

19 February 2021 01:35 PM

- Since in pipelining there is parallel execution of programs it may so happen that the second instruction requires a data which was to be updated by the first instruction which is not completed yet.
- Now the second instruction will work upon that outdated data and the hazard occurs.

Example

$A \leftarrow 3 + A$

$B \leftarrow 4 \times A$

  - We consider a =5
    After the operation b should be 4 * (3+5)=40
    But b = 4*5=20 since the first instruction has not completed yet.

- More specifically if we consider A <- B + C
  - Here A is destination
  - B and C is source
- So when a destination of previous operation is required as a source of the next operation the data hazard occurs.
- To prevent this we need to induce a stall in the system as shown.

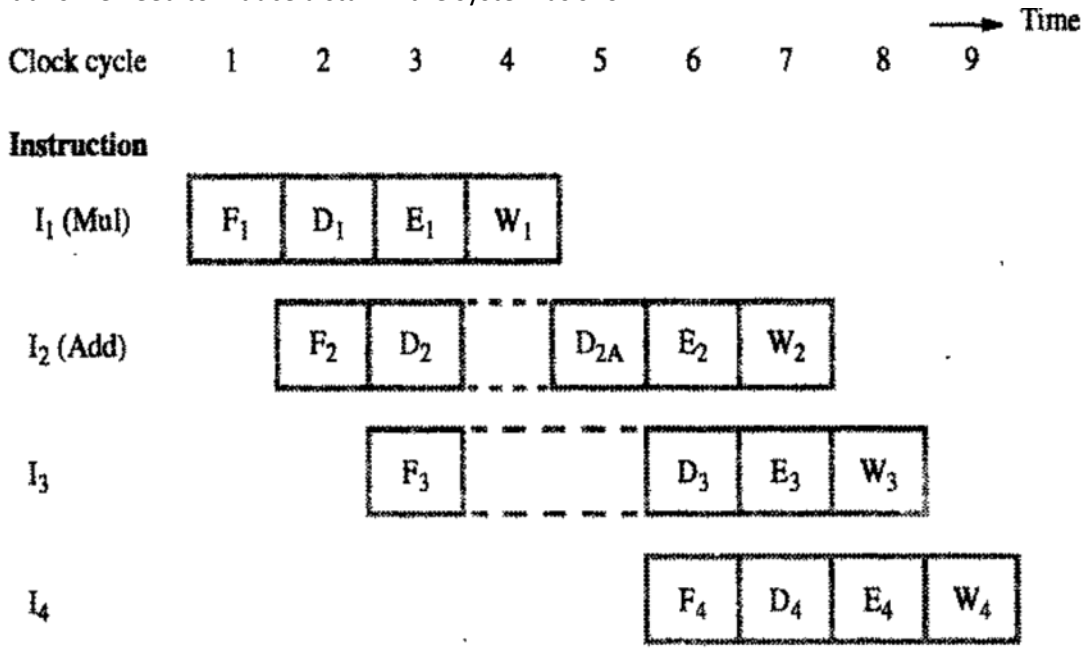


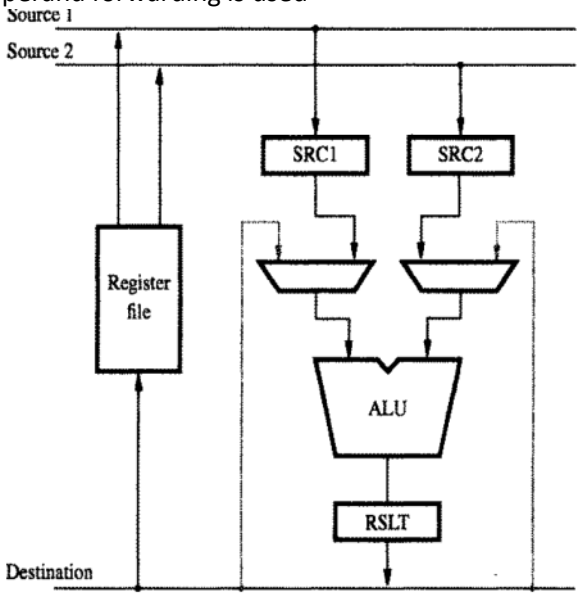
Figure 8.6 Pipeline stalled by data dependency between D_2 and W_1.

When during decoding it realized that source was destination in previous operation it stalls and again starts the decode step when the previous operation is completed resulting in stall for two clk. cycle.

Operand forwarding

19 February 2021 01:54 PM

- When the condition of data hazard is asserted
  - Operand forwarding is used



(a) Datapath

This is a part of processor showing ALU
While the result is available in the bus(while transferring to data register) the mux at src 1 is selected to take that result instead of src 1
This is called operand forwarding.

- Steps

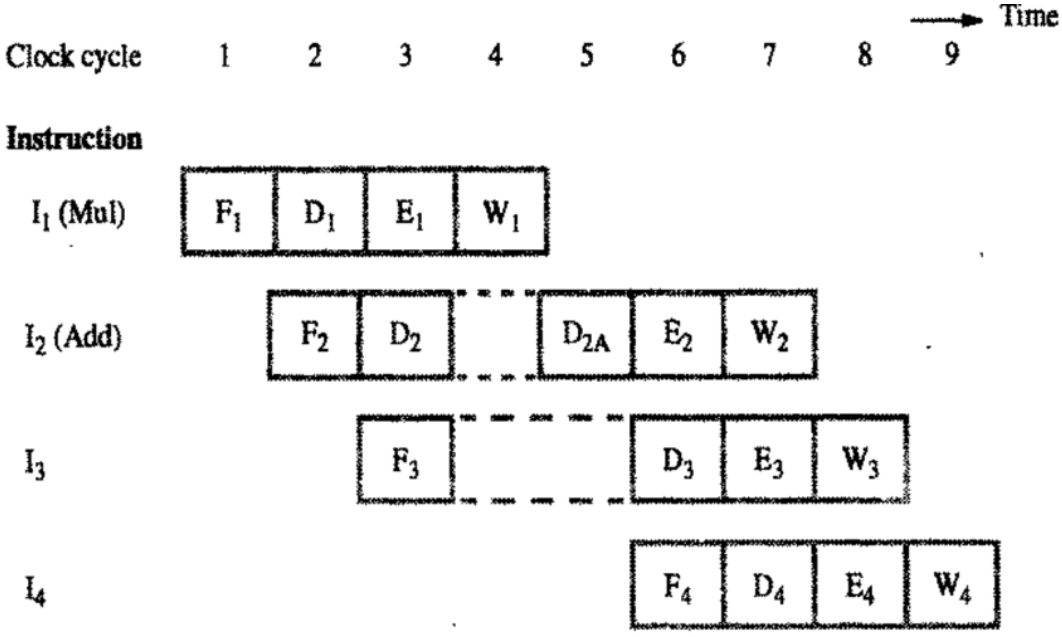


Figure 8.6 Pipeline stalled by data dependency between D_2 and W_1.

This is the figure not using operand forwarding

a. Data hazard asserted clk. 3
b. Data forwarding happens in clk.4
   Thus now the stall is of single clk.

Software handling

19 February 2021 03:37 PM

- The task of detecting and handling data hazard can also be left to software the compiler.
- For example

```
I1:  Mul   R2,R3,R4
     NOP
     NOP
I2:  Add   R5,R4,R6
```

Here the compiler detected the data dependency and introduced to no-operation stages which induces the required two clk. Stalls.

In this concept it can also be appreciated that the compiler and the architecture has close ties one thing may be handled by compiler to reduce the hardware complexity.

Side effects

19 February 2021 03:44 PM

The instruction which changes the contents of registers other than destination is affected it is called a side effect

This leads to other sorts of data dependency.

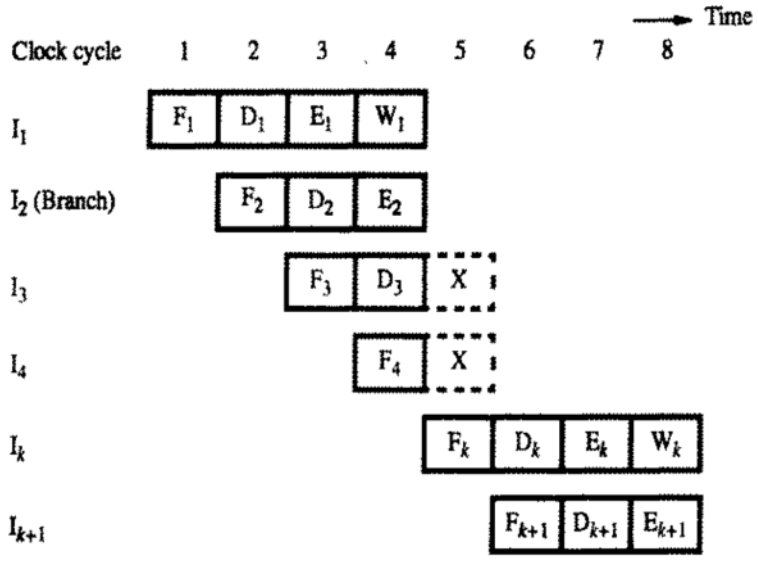
Instruction hazards

19 February 2021 03:46 PM

- The purpose of the instruction fetch unit is to constantly supply the instructions.
- When it fails for example due to cache miss it results in pipeline stall this is instruction hazard.

Unconditional branches

19 February 2021 03:55 PM



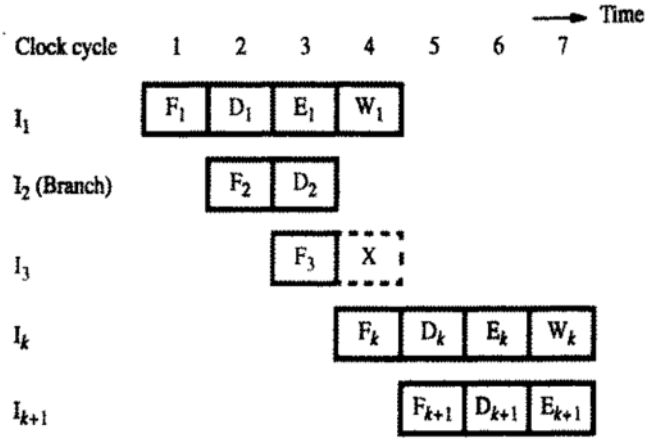
(a) Branch address computed in Execute stage

Consider the above figure

- Second instruction is the branch instruction.
- While the branch is being calculated in the execute stage of second instruction.
- The 3rd and 4th instruction is already started to be fetched but the branch is kth instruction
- As soon as the processor knows this it stops the third and forth instructions and starts kth instruction.
- This introduces a stall of two clk. Cycle in this 4 stage process.

Thus we have a branch penalty of two clk. Cycles in this example

In typical processor the decoder unit is provided with a dedicated hardware to compute branches early in decode stage this reduces the branch penalty as shown.



(b) Branch address computed in Decode stage

Instruction queue and prefetching

19 February 2021 04:57 PM

Consider the following case

- Instead of fetching instructions in a streamlined way,
- We introduced a faster and smarter fetch unit. (it may also be called a fetch-dispatch unit)
    - The fetch unit is faster than other processing unit.
    - It is able to calculate branch instructions.
- We also have an instruction queue.

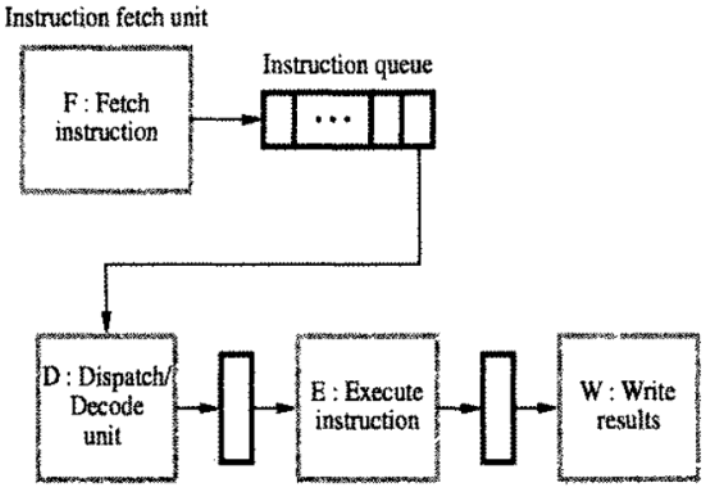


Figure 8.10 Use of an instruction queue in the hardware organization of Figure 8.2*b*.

Here after the instructions are dispatched it is again decoded, read, executed and results written

The decode unit in dispatch is only to guess next instructions.

Example:

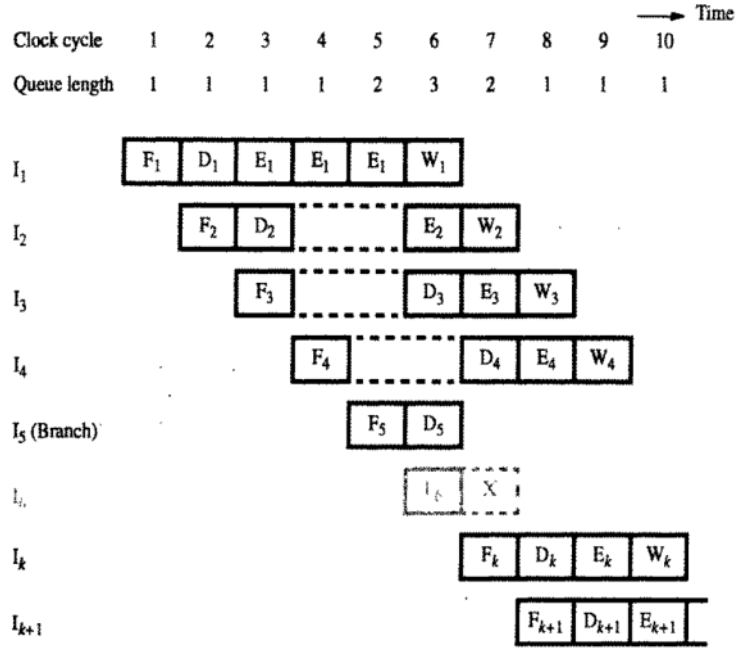


Figure 8.11 Branch timing in the presence of an instruction queue. Branch target address is computed in the D stage.

- The first instruction introduces a 2 clk. Cycle stall.
- This leads to filling the instruction queue with two instructions at clk. 5

- At 5th clock the branch instruction is fetched
- At 6th clock D_3 internal decode unit is working with I_3 and D_5 external decode unit (fetch/dispatch) works with branch instructions.
- This discards I_6 from queue and brought the kth (branched instruction) to queue.
- Thus the branch instruction does not introduces a stall
-

Points

- For this **branch folding** to occur the instruction queue must be full most of the time (instructions must be adequate supply).
- This also helps in cache miss.

Conditional branches

19 February 2021 05:21 PM

Delayed branch

Consider the following program which is reordered by the compiler as shown.

LOOP	Shift_left	R1
	Decrement	R2
	Branch=0	LOOP
NEXT	Add	R1,R3

(a) Original program loop

LOOP	Decrement	R2
	Branch=0	LOOP
	Shift_left	R1
NEXT	Add	R1,R3

(b) Reordered instructions

Figure 8.12 Reordering of instructions for a delayed branch.

This reordering is executed as follows

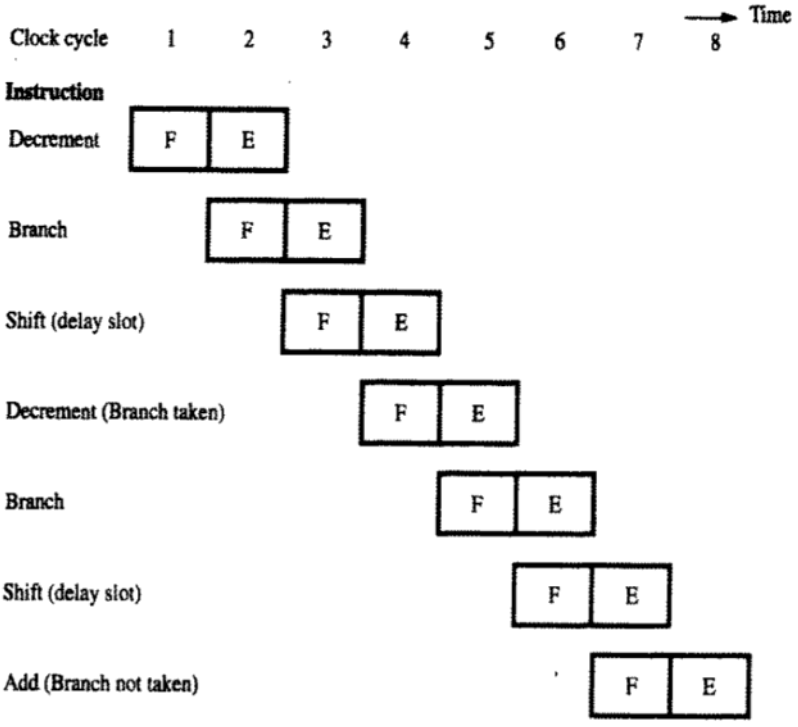


Figure 8.13 Execution timing showing the delay slot being filled during the last two passes through the loop in Figure 8.12*b*.

- When the branch instruction is executed it is ensured that the instruction next to that is also

executed fully whatever be the result of branch condition. This is called delayed branching.

- Normally(without reordering) if the branch instruction resulted in true some already executing instructions in the pipeline would have to be discarded.
- The no. of such discarded instructions is called delay slot.
- In delayed branching if no. of statements which are to be executed before branch and which if executed after it will not affect the logic is same as delay slot the pipe line will not stall at all.
- Such instruction are very difficult to find.

Branch predictions

19 February 2021 07:00 PM

Static branch predictions

- It is randomly predicted by the processor whether the condition was true or false.
- The pipeline procedure is taken accordingly.
- But further execution takes place in a speculative way.
    - Speculative execution means that processor registers or memory locations are not updated until conformation was made that the speculation was correct.

One such method is to predict the branch to be always true or always false

- A random guess this way will save 50 percent of time.

Another method is to let the compiler decide whether the branch is taken or not

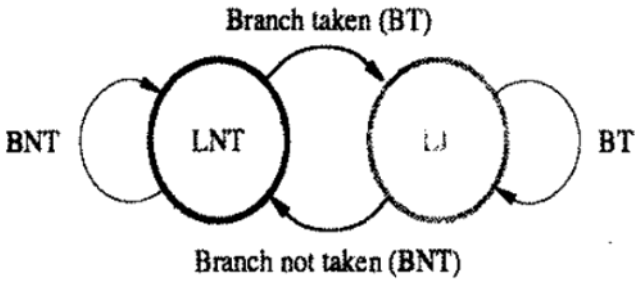
- In case of loos it is more likely that branch will be taken so compiler can set a bit to indicate that.

Both these methods will execute the same way in different run so they are static methods.

Dynamic branch pridictions

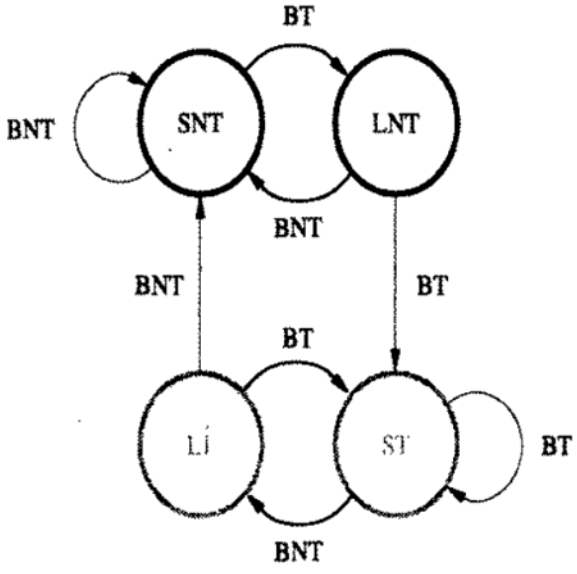
19 February 2021 07:27 PM

- In its simplest form it maintains two states for a branch
  - Likely taken LT - pipeline assumes branch is to be taken
  - Likely not taken LNT - pipeline assumes branch is not to be taken



(a) A 2-state algorithm

    The initial state may be decided by the compiler
  - This has a disadvantage in a loop though.
  - Lets say the compiler initializes to LT
  - At the last pass of the loop the loop goes to LNT state (because last pass is false)
  - Same is the case when compiler predicts LNT first
  - So if the loop is encountered again it will introduce a stall. (even if the compiler gives a prediction)
- 4 state prediction
  - This has 4 state and solves the above problem
    - Strongly Likely taken ST - pipeline assumes branch is to be taken
    - Likely taken LT - pipeline assumes branch is to be taken
    - Likely not taken LNT - pipeline assumes branch is not to be taken
    - Strongly Likely not taken SNT - pipeline assumes branch is not to be taken



(b) A 4-state algorithm

  - Now in the last step of loop the state changes from ST to LT
  - And hence in the next encounter of loop pipeline will not stall.

It is however to be noted that pipeline will at least stall once while exiting the loop.

Influence on instruction sets

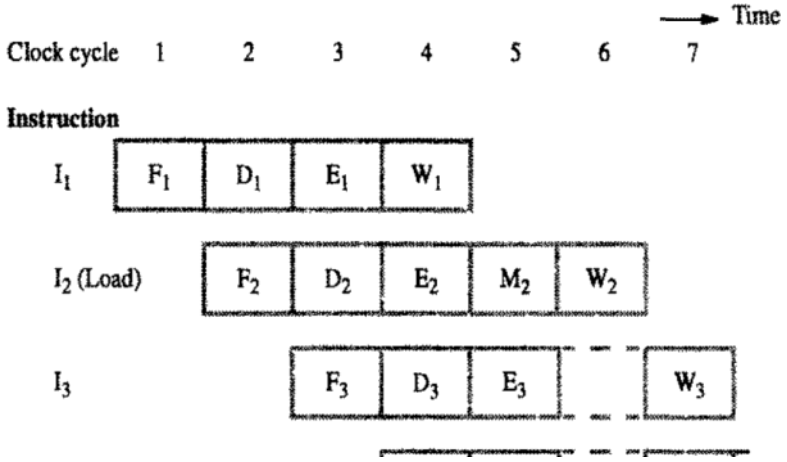
22 February 2021 08:53 AM

- Certain instructions are more suited to a pipelined execution.
- Two aspects of instruction sets
    - Addressing modes
    - condition code flags
      Are discussed here.

Addressing modes

22 February 2021 08:56 AM

Comparison of two different addressing modes in a processor
LOAD X(R1)R2



The load instruction is fetched in cycle 2
During the E_2 phase X+ [R1] is calculated
M_2 phase memory is accessed.

In total this requires 5 clk. Cycles to complete and introduces 1 clk. Cycle of stall.

LOAD (X(R1))R2

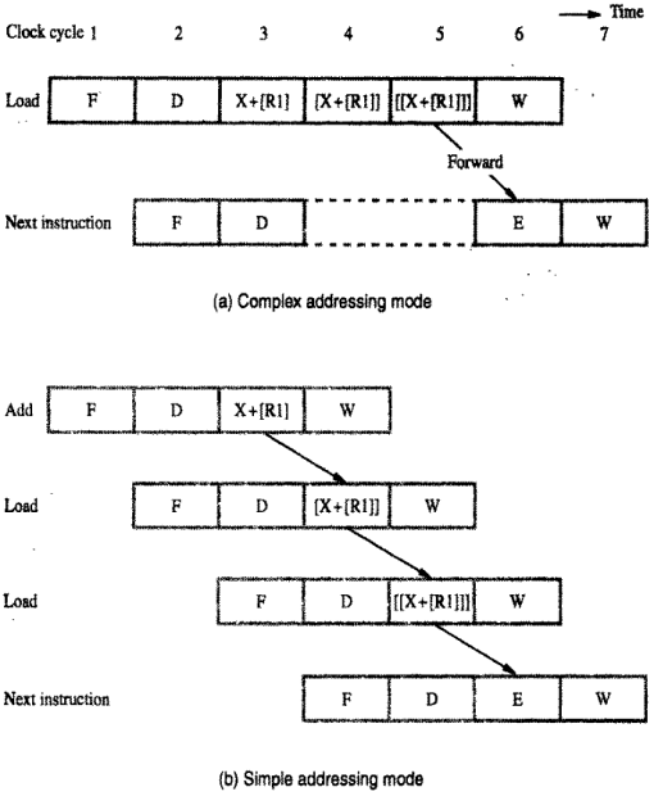


Figure 8.16 Equivalent operations using complex and simple addressing modes.

Here whether we used a complex or simple modes of instruction it takes 7 cycles to complete.

So to reduce pipeline stall certain preferred modes are to be used these are determined by the

following rules

- Access to an operand should not require more than one memory access
- Only load and store instruction accesses memory operands
- The addressing modes should not have side effects.

These features are found in
Register mode
Register indirect mode
Index mode

Condition codes

22 February 2021 09:11 AM

In a simple sequence of branch instruction the following happens

- First the condition is evaluated and matched
    - This introduces a delay and thus the whole business of branch prediction
- To reduce such delay the instructions may be rearranged so that between branch evaluation and comparing some useful instruction is carried out. (accommodating the delay)
- The instructions to be carried must not effect condition codes (because it will be used in branch)

Thus the following rules are used

- The condition code flags should be affected by as few instruction as possible
- Compiler must be able to specify which instruction actually change the condition codes.

Data path and control considerations

22 February 2021 12:05 PM

For a CPU implementing pipeline the following changes to data path and bus structure is made

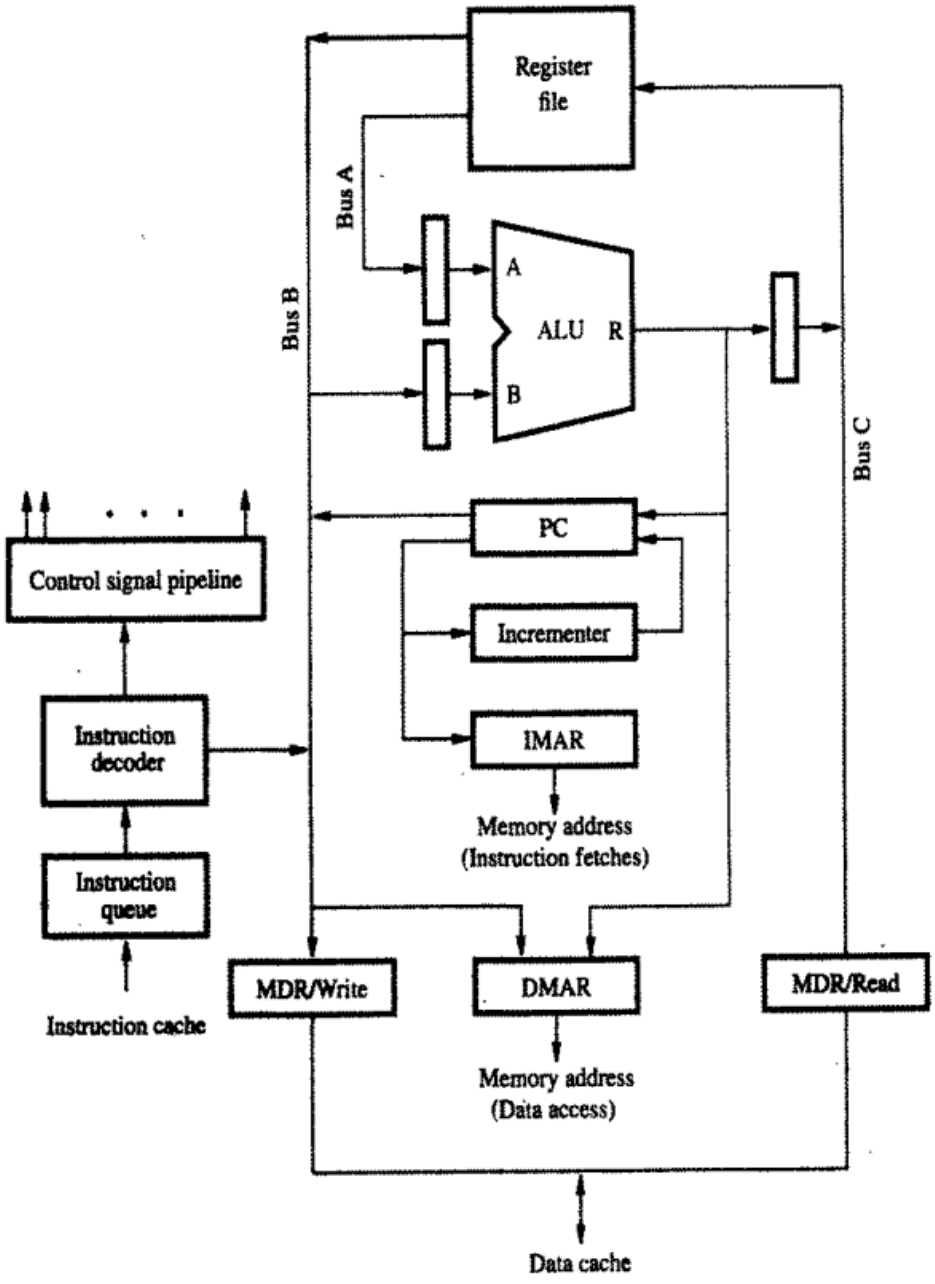


Figure 8.18 Datapath modified for pipelined execution with interstage buffers at the input and output of the ALU.

The following operations can be performed independently in the processor of Figure 8.18:

- Reading an instruction from the instruction cache
- Incrementing the PC
- Decoding an instruction
- Reading from or writing into the data cache
- Reading the contents of up to two registers from the register file
- Writing into one register in the register file
- Performing an ALU operation

Omissions

22 February 2021 12:10 PM

8.6 and 8.7 from carl hamsher
8.8 of carl hamsher deals with performance and some what important