MCA

Problem Solving & Programming

Program to find the size of different basic data types in C

```
#include<stdio.h>
#include<conio.h>
void main()
{
int x;
float y;
char z;
double d;
long l;
clrscr();
printf("size of integer= %d",sizeof(x));
printf("\nsize of float= %d",sizeof(y));
printf("\nsize of char= %d",sizeof(z));
printf("\nsize of double= %d",sizeof(d));
printf("\nsize of long= %d",sizeof(l));
getch();
}
```

Program in C for arithmetic operations between two integers. Program should guide users with proper message/menu on the console.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num1,num2,sum,difference,mul,choice;
float div;
start :
clrscr();
printf("\n1.addition");
printf("\n2.subtraction");
printf("\n3.multiplication");
printf("\n4.division");
printf("\n5.exit.");
printf("\n\nenter your choice:");
scanf("%d",&choice);
switch(choice)
{
case 1:
printf("\nenter first number");
scanf("%d",&num1);
printf("\nenter second number");
scanf("%d",&num2);
printf("\n sum of two numbers=%d",(num1+num2));
break;
case 2:
printf("\nenter first number");
scanf("%d",&num1); printf("\nenter second number");
scanf("%d",&num2);
printf("\n difference of two numbers=%d",(num1-num2));
break;
case 3:
printf("\nenter first number");
scanf("%d",&num1);
printf("\nenter second number");
scanf("%d",&num2);
printf("\n product of two numbers=%d",(num1*num2));
break;
case 4:
printf("\nenter first number");
scanf("%d",&num1);
printf("\nenter second number");
scanf("%d",&num2);
printf("\n division of two numbers=%d",(num1/num2));
break;
case 5:
printf("\npress enter to exit");
getch();
exit(0);
default :
printf("\nyou have entered wrong choice:\n please enter new choice:");
goto start;
}
}
```

A function Pali(Sting S) to find whether S is palindrome or not.

```
pali (char s[])
{
int length,i,j,flag=0;
length=strlen(s);
for(i=0,j=length-1;i<(length/2),j>(length/2);i++,j--)
{
if(s[i]!=s[j])
flag=1;
}
if(flag==1)
{
printf("\nstring is not pallendrome");
}
else
{
printf("\nstring is pallendrome");
}
getch();
}
```

C program to print this triangle:

```
*
***
*****
*******
*********
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j;
for(i=0;i<12;i+=2)
{
for(j=i;j>=0;j--)
{
printf("*");
}
printf("\n");
}
getch();
}
```

C program to compute the average marks in a subject of all the students in a class

```
#include<stdio.h>
#include<conio.h>
void main()
{
int marks, i, noofstudents, sum=0;
float avg;
clrscr();
printf("enter no of students");
scanf("%d",&noofstudents);
printf(„\n enter marks of students");
for(i=0;i<noofstudents;i++);
{
scanf("%d",&marks);
sum+=marks;
}
printf("\naverage of class=%f",(sum/noofstudents));
getch();
}
```

Explain pointer arithmetic with example. Also explain advantages of *malloc* and *calloc*

A pointer is a variable that contains the memory location of another variable. The syntax is as shown below. You start by specifying the type of data stored in the location identified by the pointer. The asterisk tells the compiler that you are creating a pointer variable. Finally you give the name of thevariable.
type * variable name **Example:**
int *ptr; float *string;
C allows us to add integers to or subtract integers from pointers as well as to subtract one pointer from the other. We can also use short hand operators with the pointers p1+=; sum+=*p2; etc.,
we can also compare pointers by using relational operators the expressions such as p1 >p2 , p1==p2 and p1!=p2 are allowed. /*Program to illustrate the pointer expression and pointer arithmetic*/ #include< stdio.h > main() { int ptr1,ptr2; int a,b,x,y,z; a=30;b=6; ptr1=&a; ptr2=&b; x=*ptr1+ *ptr2 –6; y=6*- *ptr1/ *ptr2 +30; printf("\nAddress of a +%u",ptr1);

printf("\nAddress of b %u",ptr2); printf("\na=%d, b=%d",a,b); printf("\nx=%d,y=%d",x,y); ptr1=ptr1 + 70; ptr2= ptr2; printf("\na=%d, b=%d",a,b); }.

Advantage of Malloc and Calloc

malloc() allocates byte of memory, whereas calloc()allocates block of memory.

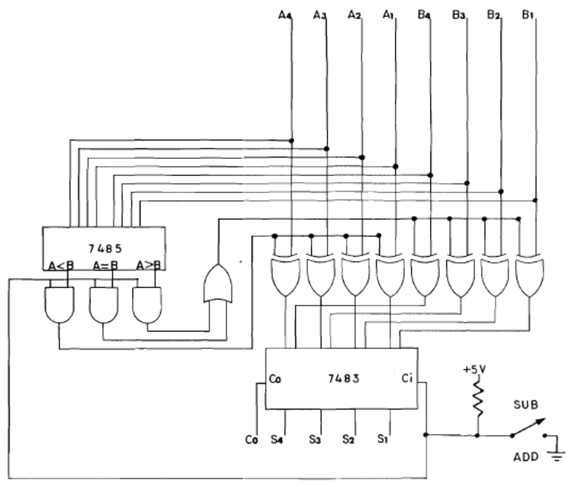
Calloc(m, n) is essentially equivalent to p = m*malloc(n); memset(p, 0, m * n); The zero fill is all-bits-zero, and does not therefore guarantee useful null pointer values (see section 5 of this list) or floating-point zero values. Free is properly used to free the memory allocated by calloc.

Malloc(s); returns a pointer for enough storage for an object of s bytes. Calloc(n,s); returns a pointer for enough contiguous storage for n objects, each of s bytes. The storage is all initialized to zeros.

Simply, malloc takes a single argument and allocates bytes of memory as per the argument taken during its invocation. Where as calloc takes two aguments, they are the number of variables to be created and the capacity of each vaiable (i.e. the bytes per variable).

Computer Organisation ans Assembly Language Programming

A logic circuit that accepts a four digit binary input and creates an odd parity bit, a sign check bit and a more than two zero value test bit. The odd parity bit is created for the four bit data. The sign bit is set to 1 if the left most bit of the data is 1. Zero value bit is set to 1 if three of the input bits are zero. The truth table and use K-map to design the Boolean expressions for each of the output bits. The resulting circuit diagram using AND – OR – NOT gates.

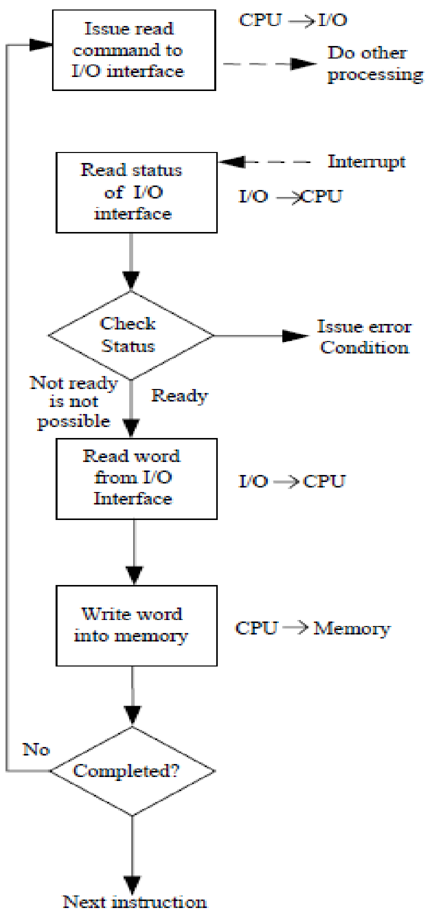


Explain how the I/O will be performed if

(i) Interrupt Driven Input/ Output Scheme is used.

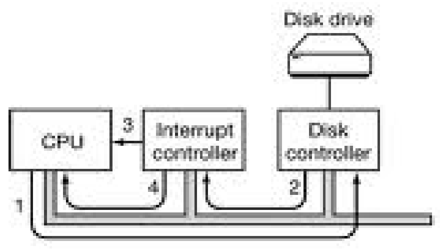
With interrupt driven I/O, when the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the processor stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing which results in the waiting time by the processor being reduced

.

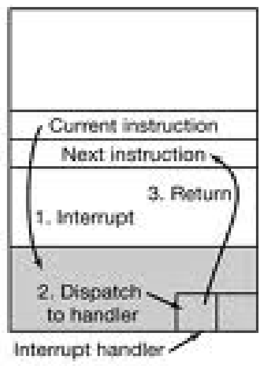


The interrupt-driven I/O mechanism for transferring a block of data is shown in Figure. Please note that after issuing a read command (for input) the CPU goes off to do other useful work while I/O interface proceeds to read data from the associated device. On the completion of an instruction cycle, the CPU checks for interrupts (which will occur when data is in data register of I/O interface and it now needs CPU's attention). Now CPU saves the important register and processor status of the executing program in a stack and requests the I/O device to provide its data, which is placed on the data bus by the I/O device. After taking the required action with the data, the CPU can go back to the program it was executing before the interrupt.

Interrupt-Processing The occurrence of an interrupt fires a numbers of events, both in the processor hardware and software. Figure 8 shows a sequence



(a)



(b)

When an I/O device completes an I/O operation, the following sequence of hardware events occurs: 1. The device issues an interrupt signal to the processor. 2. The processor finishes execution of the current instruction before responding to the interrupt. 3. The processor tests for the interrupts and sends an acknowledgement signal to the device that issued the interrupt.

When an I/O device completes an I/O operation, the following sequence of hardware events occurs: 1. The device issues an interrupt signal to the processor. 2. The processor finishes execution of the current instruction before responding to the interrupt. 3. The processor tests for the interrupts and sends an acknowledgement signal to the device that issued the interrupt.

4. The minimum information required to be stored for the task being currently executed, before the CPU starts executing the interrupt routine (using its registers) are: (a) The status of the processor, which is contained in the register called program status word (PSW), and (b) The location of the next instruction to be executed, of the currently executing program, which is contained in the program counter (PC).

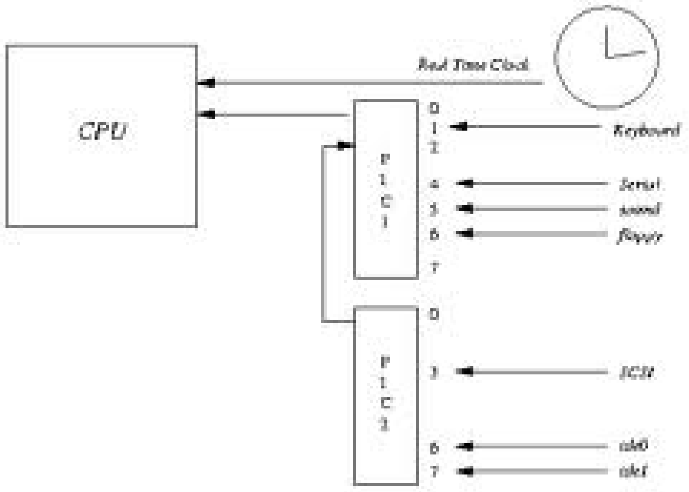
5. The processor now loads the PC with the entry location of the interrupt-handling program that will respond to this interrupting condition. Once the PC has been loaded, the processor proceeds to execute the next instruction, that is the next instruction cycle, which begins with an instruction fetch. Because the instruction fetch is determined by the contents of the PC, the result is that control is transferred to the interrupt-handler program. The execution results in the following operations:

6. The PC & PSW relating to the interrupted program have already been saved on the system stack. In addition, the contents of the processor registers are also needed to be saved on the stack that are used by the called Interrupt Servicing Routine because these registers may be modified by the interrupt-handler. Figure shows a simple example. Here a user program is interrupted after the instruction at location N. The contents of all of the registers plus the address of the next instruction (N+1) are pushed on to the stack.

7. The interrupt handler next processes the interrupt. This includes determining of the event that caused the interrupt and also the status information relating to the I/O operation.

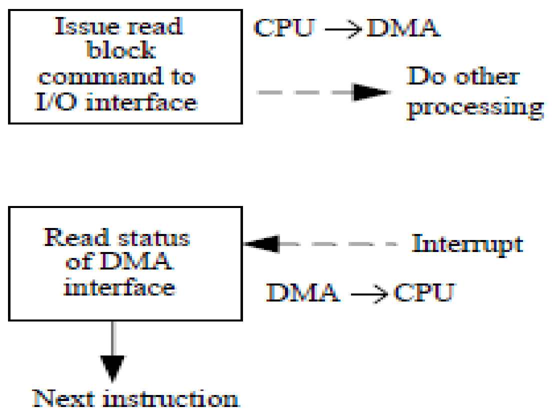
8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers, which are shown in Figure .

9. The final step is to restore the values of PSW and PC from the stack. As a result, the instruction to be executed will be from the previously interrupted program.



Direct Memory Access is used.

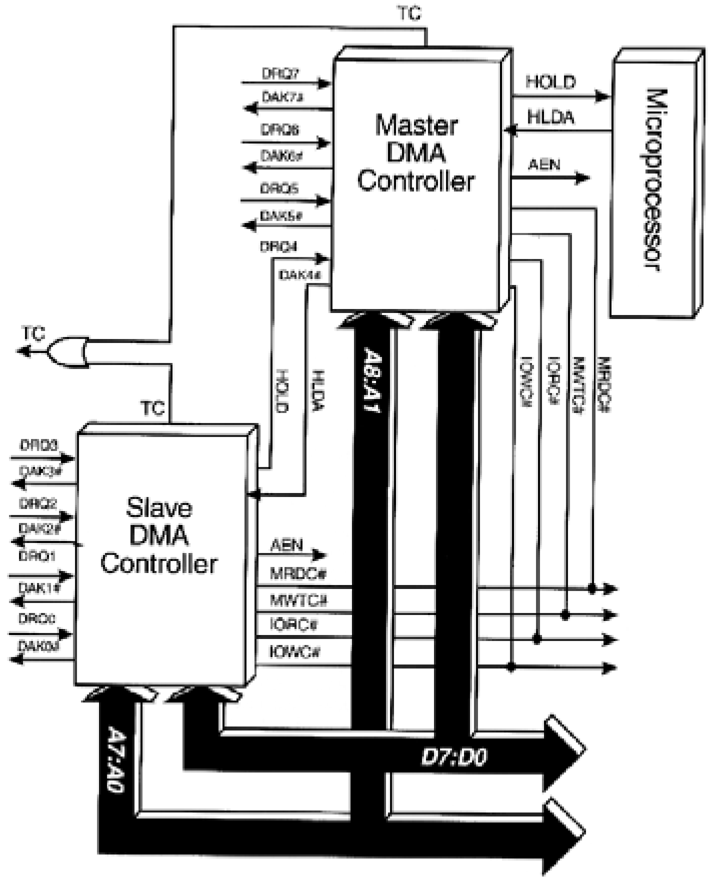
In both interrupt-driven and programmed I/O, the processor is busy with executing input/output instructions and the I/O transfer rate is limited by the speed with which the processor can test and service a device. What about a technique that requires minimal intervention of the CPU for input/output? These two types of drawbacks can be overcome with a more efficient technique known as DMA, which acts as if it has taken over control from the processor. Hence, the question is: why do we use DMA interface? It is used primarily when a large amount of data is to be transferred from the I/O device to the Memory.


Issue read block command to I/O interface
CPU → DMA
Do other processing
Read status of DMA interface
Interrupt
DMA → CPU
Next instruction

DMA Function Although the CPU intervention in DMA is minimised, yet it must use the path between interfaces that is the system bus. Thus, DMA involves an additional interface on the system bus. A technique called cycle stealing allows the DMA interface to transfer one data word at a time, after which it must return control of the bus to the processor. The processor merely delays its operation for one memory cycle to allow the directly memory I/O transfer to “steal” one memory cycle. When an I/O is requested, the processor issues a command to the DMA interface by sending to the DMA interface the following information (Figure 10): • Which operations (read or write) to be performed, using the read or write control lines. • The address of I/O devices, which is to be used, communicated on the data lines. • The starting location on the memory where the information will be read or written to be communicated on the data lines and is stored by the DMA interface in its address register. • The number of words to be read or written is communicated on the data lines and is stored in the data count register.

The DMA interface transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor. When the transfer is complete, the DMA interface sends an interrupt signal to the processor. Thus, in DMA the processor involvement can be restricted at the beginning and end of the transfer, which can be shown as in the figure above. But the question is when should the DMA take control of the bus?

For this we will recall the phenomenon of execution of an instruction by the processor. Figure 11 below shows the five cycles for an instruction execution. The Figure also shows the five points where a DMA request can be responded to and a point where the interrupt request can be responded to. Please note that an interrupt request is acknowledged only at one point of an instruction cycle, and that is at the interrupt cycle.



Useful Terms

DIMM :-

Dual Inline Memory Module or DIMM is a series of Random Access Memory (RAM) chips mounted on a small printed circuit board. The entire circuit collectively forms a memory module. DIMMs are commonly used in personal computers, servers and high-end workstations. The DIMM makes physical contact with the data bus of the computer through teeth like connectors that fit into a socket on the mother board. DIMMs use a 64-bit data path, since processors used in personal computers including the Intel Pentium have a 64-bit data width.

LCD monitors :-

LIQUID CRYSTAL DISPLAYS (LCD) LCDs are the screens of choice for portable computers and lightweight screens. They consume very little electricity and have advanced technologically to quite good resolutions and colour support. They were developed by the company RCA in the 1960s. LCDs function simply by blocking available light so as to render display patterns.

Core of a processor :-

The "core" in a processor is the microprocessor inside of the CPU (Central Processing Unit). It is the part of the processor that actually performs the reading and executing of instructions. For example, if we have a Dual Core CPU then we have 2 microprocessors inside of the CPU, this allows us to do two things at once, as a microprocessor can only do one thing at a time the only exception to this is Multi-Threading which allows one core to do the work of multiple cores, but its not as fast as having multiple cores.

SATA :-

Serial ATA (**SATA** or **Serial Advanced Technology Attachment**) is a computer bus interface for connecting host bus adapters to mass storage devices such as hard disk drives and optical drives. Serial ATA was designed to replace the older ATA (AT Attachment) standard (also known as EIDE), offering several advantages over the older parallel ATA (PATA) interface: reduced cable-bulk and cost (7 conductors versus 40), native hot swapping, faster data transfer through higher signaling rates, and more efficient transfer through an (optional) I/O queuing protocol.

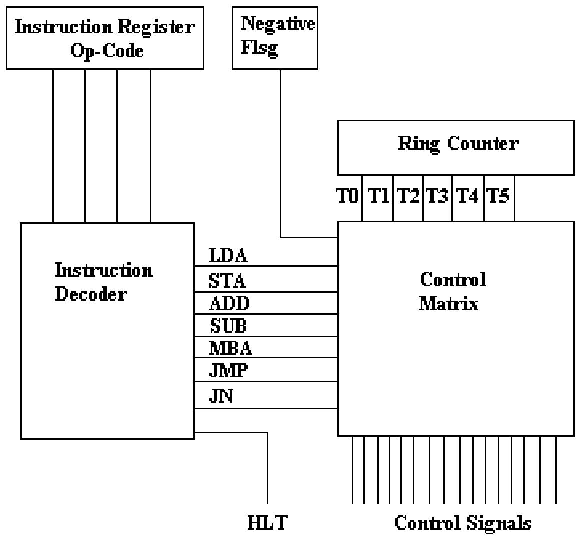
SATA host-adapters and devices communicate via a high-speed serial cable over two pairs of conductors. In contrast, parallel ATA (the redesignation for the legacy ATA specifications) used a 16-bit wide data bus with many additional support and control signals, all operating at much lower frequency. To ensure backward compatibility with legacy ATA software and applications, SATA uses the same basic ATA and ATAPI command-set as legacy ATA devices.

RAID level 5 :-

This level belongs to independent access category. Its main features are: a) Employs independent access as that of level 4 and distributes the parity strips across all disks. b) The distribution of parity strips across all drives avoids the potential input/output bottleneck found in level 4. I/O Request Rate Read is Excellent while write is fair Data Transfer Rate Read is fair while write is poor It is used in High request rate read intensive, data lookup applications

Zone Bit Recording (**ZBR**) is used by disk drives to store more sectors per track on outer tracks than on inner tracks. It is also called **Zone Constant Angular Velocity** (**Zone CAV** or **Z-CAV** or **ZCAV**). On a disk consisting of concentric tracks, the physical track length may or may not be increased with distance from the center hub. Therefore, holding storage density constant, the track storage capacity likewise increases with distance from the center. ZBR is a compromise between CLV (which packs the most bits onto a disk, but has very slow seek times) and CAV (which has faster seek times, but stores fewer bits on a disk). Hard disk controllers implement ZBR by varying the rate at which it reads and writes - faster on outer tracks. Some other ZBR drives, such as the 3.5" floppy drives in the Apple IIGS and older Macintosh computers, spin the medium faster when reading or writing inner tracks. One side effect of ZBR is the raw data transfer rate of the disk when reading the outside tracks is much higher -- in some disks, about double -- the data transfer rate of the same disk when reading the "inner" (closest to the hub) tracks.

A variety of techniques have been used to organize a control unit. Most of them fall into two major categories: 1. Hardwired control organization 2. Micro programmed control organization. In the hardwired organization, the control unit is designed as a combinational circuit. That is, the control unit is implemented by gates, flip-flops, decoder and other digital circuits. Hardwired control units can be optimized for fast operations.



The block diagram of control unit is shown in Figure . The major inputs to the circuit are instruction register, the clock, and the flags. The control unit uses the opcode of instruction stored in the IR register to perform different actions for different instructions. The control unit logic has a unique logic input for each opcode. This simplifies the control logic. This control line selection can be performed by a decoder. A decoder will have n binary inputs and 2n binary outputs. Each of these 2n different input patterns will activate a single unique output line. The clock portion of the control unit issues a repetitive sequence of pulses for the SS duration of micro-operation(s). These timing signals control the sequence of execution of instruction and determine what control signal needs to applied at what time for instruction execution.

An **instruction pipeline** is a technique used in the design of computers and other digital electronic devices to increase their instruction throughput (the number of instructions that can be executed in a unit of time). The fundamental idea is to split the processing of a computer instruction into a series of independent steps, with storage at the end of each step. This allows the computer's control circuitry to issue instructions at the processing rate of the slowest step, which is much faster than the time needed to perform all steps at once. The term pipeline refers to the fact that each step is carrying data at once (like water), and each step is connected to the next (like the links of a pipe.) The origin of pipelining is thought to be either the ILLIAC II project or the IBM Stretch project though a simple version was used earlier in the Z1 in 1939 and the Z3 in 1941 ***Disadvantages of Pipelining****:*

1. A non-pipelined processor executes only a single instruction at a time. This prevents branch delays (in effect, every branch is delayed) and problems with serial instructions being executed concurrently. Consequently the design is simpler and cheaper to manufacture.

2. The instruction latency in a non-pipelined processor is slightly lower than in a pipelined equivalent. This is because extra flip flops must be added to the data path of a pipelined processor.

3. A non-pipelined processor will have a stable instruction bandwidth. The performance of a pipelined processor is much harder to predict and may vary more widely between different programs.

Optimization of Pipelining in RISC Processors RISC machines can employ a very efficient pipeline scheme because of the simple and regular instructions. Like all other instruction pipelines RISC pipeline suffer from the problems of data dependencies and branching instructions. RISC optimizes this problem by using a technique called delayed branching. One of the common techniques used to avoid branch penalty is to pre-fetch the branch destination also. RISC follows a branch optimization technique called delayed jump as shown in the example given below:

Load $R_A \leftarrow M(A)$	F	E	D					
Load $R_B \leftarrow M(B)$		F	E	D				
Add $R_C \leftarrow R_A + R_B$			F	E				
Sub $R_D \leftarrow R_A - R_B$				F	E			
If $R_D < 0$ Return					F	E		
Stor $R_C \rightarrow M(C)$						F	E	D
Return							F	E

(a) The instruction "If $R_D < 0$ Return" may cause pipeline to empty

Load $R_A \leftarrow M(A)$	F	E	D							
Load $R_B \leftarrow M(B)$		F	E	D						
Add $R_C \leftarrow R_A + R_B$			F	E						
Sub $R_D \leftarrow R_A - R_B$				F	E					
If $R_D < 0$ Return					F	E				
NO Operation						F	E			
Stor $R_C \rightarrow M(C)$ Or Return as the case may be								F	E	D
Return									F	E

(b) The No operation instruction causes decision of the If instruction known, thus correct instruction can be fetched.

Load $R_A \leftarrow M(A)$	F	E	D					
Load $R_B \leftarrow M(B)$		F	E	D				
Sub $R_D \leftarrow R_A - R_B$			F	E				
If $R_D < 0$ Return				F	E			
Add $R_C \leftarrow R_A + R_B$					F	E		
Stor $R_C \rightarrow M(C)$						F	E	D
Return							F	E

(c) The branch is calculated before, thus the pipeline need not be emptied. This is delayed branch.

Figure 9: Delayed Branch