Problems

1. If the 1997 version of a computer executes the code fragment

   FOR i=1 TO 10000 DO {A[i] ← B[i].C[i]; A[i] ←A[i] + D[i]}

   in 200s and the version of the computer made in the year 2000 execute the same code in 130s. It takes 5,000 cycles to execute the code on the first version and 3,000 cycles to execute the same code on the second version.
   a. What is the speedup that the manufacturer has achieved over the three-year period?
   b. What are that system's CPI and IPC values for the program?
   c. What is your comment on using these 2 metrics to measure a system performance?

Solution (Hint: see Lecture 8):

a. Speedup = Execution $time_{before}$ / Execution $time_{after}$ = 200s / 130s = 1.5385. Clearly, this manufacturer is falling well short of the industrywide performance growth rate.

b. CPI = #Cycles / #Instructions

The 2-instruction loop is executed 10,000 times, so the total number of instructions executed is 10,000x2 = 20,000. Therefore, on the first version

CPI = 5,000/20,000 = 0.25
IPC = 1/CPI = 4

On the second version

CPI = 3,000/20,000 = 0.15
IPC = 1/CPI = 0.067

c. **Comment:** When using IPC and CPI to compare systems, it is important to remember that high IPC values indicate that the reference program took fewer cycles to execute than low CPI values. Thus, a large IPC tends to indicate good performance; while a large CPI indicates poor performance.

2. Suppose a computer spends 90 percent of its time handling a particular type of computation when running a given program, and its manufacturers make a change that improves its performance on that type of computation by a factor of 10.
   a. If the program originally took 100s to execute, what will its execution time be after the change?
   b. If the program originally took 110s to execute, and the manufacturer made a change that improves the computer's performance on that type of computation, and if the new execution time after the improvement is 15s, what should be the improvement factor (i.e. used speedup)?
   c. What is the speedup from the old system to the new system?

Solution (Hint: see Lecture 8):

a. This is a direct application of Amdahl's Law:

$$\text{Execution Time}_{new} = \text{Execution Time}_{old} \times [\text{Frac}_{unused} + \text{Frac}_{used}/\text{Speedup}_{used}]$$

$\text{Execution Time}_{old} = 100s$, $\text{Frac}_{used} = 0.9$, $\text{Frac}_{unused} = 0.1$, and $\text{Speedup}_{used} = 10$.

This gives

$$\text{Execution Time}_{new} = 100 \times [0.1 + 0.9/10] = 19s$$

b. Again, this is a direct application of Amdahl's Law:

$$\text{Execution Time}_{new} = \text{Execution Time}_{old} \times [\text{Frac}_{unused} + \text{Frac}_{used}/\text{Speedup}_{used}]$$

$\text{Execution Time}_{old} = 110s$, **$\text{Execution Time}_{new}$** $= 15s$, $\text{Frac}_{used} = 0.9$, $\text{Frac}_{unused} = 0.1$

This gives

$$15 = 110 \times [0.1 + 0.9/ \text{Speedup}_{used}]$$

Solve this equation for Speedup_{used} gives

$$\text{Speedup}_{used} = 24.75$$

c. Using the definition of speedup, we get a speedup of 5.3 (100/19). Alternately, we could substitute the values from (a) into the speedup version of Amdahl's Law to get the same result.

3. Which improvement gives a greater reduction in execution time: One that is used 20% of time but improves performance by a factor of 2 when used, or one that is used 70% of time but only improves performance by a factor of 2.14 when used?

Solution (Hint: see Lecture 8):

Applying Amdahl's Law, we get the following equation for the first improvement:

$$\text{Speedup} = \text{Execution Time}_{old} / \text{Execution Time}_{new} = 1/[\text{Frac}_{unused} + \text{Frac}_{used}/\text{Speedup}_{used}]$$

$$\text{Execution Time}_{new} = \text{Execution Time}_{old} \times [\text{Frac}_{unused} + \text{Frac}_{used}/\text{Speedup}_{used}]$$

$$\text{Execution Time}_{new} = \text{Execution Time}_{old} \times [0.8 + 0.2/2]$$

$$\text{Execution Time}_{new} = 0.9 \times \text{Execution Time}_{old}$$

So the execution time with the first improvement is 90 percent of the execution time without the improvement. Plugging the values for the second improvement into Amdahl's Law leads to:

$$\text{Execution Time}_{new} = \text{Execution Time}_{old} \times [0.7+ 0.3/2.14]$$

$$\text{Execution Time}_{new} = 0.84 \times \text{Execution Time}_{old}$$

It shows that the execution time with the second improvement is 84 percent of the execution time without the improvement. Thus, the second improvement will have a greater impact on overall execution time despite the fact that it gives less of an improvement when it is in use.

4.

a. If the 1998 version of a computer executes a program in 200 s and the version of the computer made in the year 2000 execute the same program in 150 s, what is the speedup that the manufacture has achieved over the two-year period?

b. To achieve a speedup of 4 on a program that originally took 80 s to execute, what must be the execution time of the program be reduced to?

c. A computer architect is designing the memory system for the next version of a processor. If the current version of the processor spends 40 percent of its time processing memory references, by how much must the architect speed up the new memory system to achieve an overall speedup of 1.2? A speedup of 1.6?

Solutions (Hint: see Lecture 8):

a. Speedup = Execution $time_{before}$ / Execution $time_{after}$ = 200/150 = 1.33

b. Speedup = Execution $time_{before}$ / Execution $time_{after}$ → 4 = 80/ Execution $time_{after}$

→ Execution $time_{after}$ =80/4 = 20s

c. This is a direct application of Amdahl's Law:
Speedup = Execution $Time_{old}$ / Execution $Time_{new}$ =
= 1/[$Frac_{unused}$ + $Frac_{used}$/$Speedup_{used}$] →
For an overall speedup of 1.2, the architect must speed up the new memory system by:

$$1.2 = 1/[0.6 + 0.4/Speedup_{used}]$$
$$Speedup_{used} = 1.714$$

For an overall speedup of 1.6, the architect must speed up the new memory system by:

$$1.2 = 1/[0.6 + 0.4/Speedup_{used}]$$
$$Speedup_{used} = 16$$

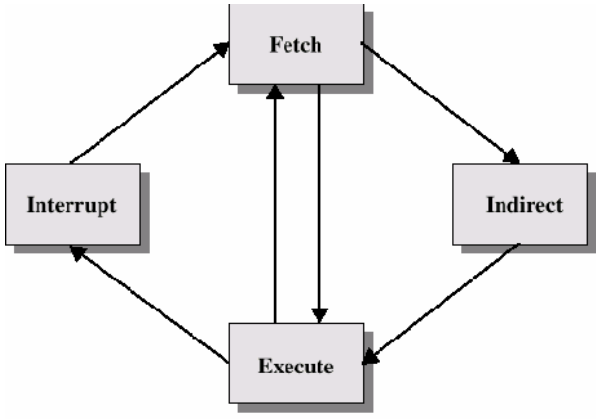
Here we again see the diminishing returns that come from repeatedly improving only one aspect of a system's performance. To increase the overall speedup from 1.2 to 1.6, we have to increase $Speedup_{used}$ by almost a factor of 10, because the 60 percent of the time that the memory system is not in use begins to dominate the overall performance as we improve the memory system performance.

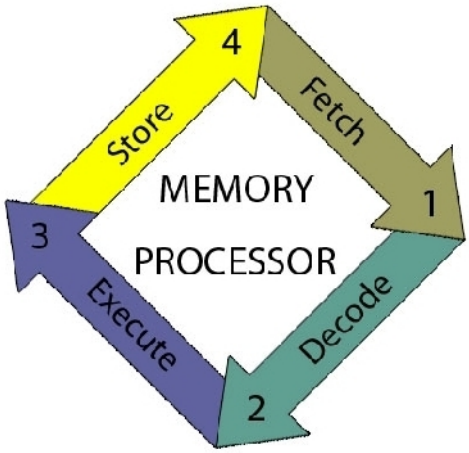
5. Describe all possible steps involved in the process of an instruction execution (i.e. Fetch, indirect, interrupt and execute cycle). You should explain the meaning of the grouping of microoperations involved in each cycle.

Solution (Hint: see Lectures 15, 16, and 24):

An **instruction cycle** (also called **fetch-and-execute cycle**, **fetch-decode-execute cycle** (**FDX**) is the time period during which a computer reads and processes a machine language instruction from its memory or the sequence of actions that the central processing unit (CPU) performs to execute each machine code instruction in a program. In other words, an instruction cycle is the fundamental sequence of steps that a CPU performs. Also known as the "fetch-execute cycle," it is the time in which a single instruction is fetched from memory, decoded and executed. The first half of the cycle transfers the instruction from memory to the instruction register and decodes it. The second half executes the instruction

An **instruction cycle** is the basic operation cycle of a computer. It is the process by which a computer retrieves a program instruction from its memory, determines what actions the instruction requires, and carries out those actions. This cycle is repeated continuously by the CPU, from bootup to when the computer is shut down. This cycle is repeated continuously by the central processing unit (CPU), from bootup to when the computer is shut down.





Circuits Used

The circuits used in the CPU during the Fetch-Execute cycle are:

- **Program counter (PC)** - an incrementing counter that keeps track of the memory address of the instruction that is to be executed next
- **Memory address register (MAR)** - holds the address of a memory block to be read from or written to
- **Memory data register (MDR)** - a two-way register that holds data fetched from memory (and ready for the CPU to process) or data waiting to be stored in memory
- **Instruction register (IR)** - a temporary holding ground for the instruction that has just been fetched from memory
- **Control unit (CU)** - decodes the program instruction in the IR, selecting machine resources such as a data source register and a particular arithmetic operation, and coordinates activation of those resources
- **Arithmetic logic unit (ALU)** - performs mathematical and logical operations

The Fetch-Execute cycle in Transfer Notation
Expressed in register transfer notation:

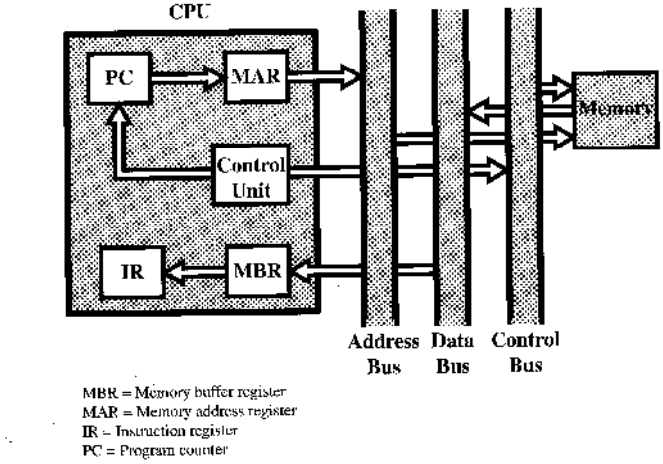
The Fetch Cycle

The simple fetch cycle actually consists of three steps and four microoperations. Each microoperation involves the movement of data into or out of a register. So long as these movements do not interfere with one another, several of them can take place during one step, saving time. At the beginning of the fetch cycle, the address of the next instruction to be executed is in the program counter PC. The first step is move that address to the memory address register MAR, because this is the only

register connected to the address lines of the system bus. The second step is to bring in the instruction. The desired address (in the MAR) is placed on the address bus, the control unit issues a READ command on the control bus, and the result appears on the data bus and is copied into the memory buffer register MBR. WE also need to increment the PC by 1 to get ready for the next instruction. Because these two actions (read word from memory, add 1 to PC) do not interfere with each other, we can do them simultaneously to save time. The third step is to move the contents of the MBR to the instruction register IR. This frees up the MBR for use during a possible indirect cycle. Thus, the simple fetch cycle actually consists of three steps and four microoperations.

As we mentioned above, we begin by looking at the fetch cycle, which occurs at the beginning of each instruction cycle and causes an instruction to be fetched from memory. For purposes of discussion, we assume the organization depicted in the figure below. Four registers are involved:

- **Memory address register (MAR):** Is connected to the address lines of the system bus. It specifies the address in memory for a read or write operation.
- **Memory buffer register (MBR):** Is connected to the data lines of the system bus. It contains the value to be stored in memory or the last value read from memory.
- **Program counter (PC):** Holds the address of the next instruction to be fetched.
- **Instruction register (IR):** Holds the last instruction fetched.



Data Flow, Fetch Cycle

In the fetch cycle, each microoperation involves the movement of data into or out of a register. So long as these movements do not interfere with one another, several of them can take place during one step, saving time. Symbolically, we can write this sequence of events as follows:

t_1: MAR ←[PC] //Move contents of PC to MAR.
t_2: MBR ←Memory //Move memory location specified by MAR to MBR.
PC ←[PC]+1 //Increment the PC for the next cycle at the same time
t_3: IR ←[MBR] //Move contents of memory location specified by MAR to IR.

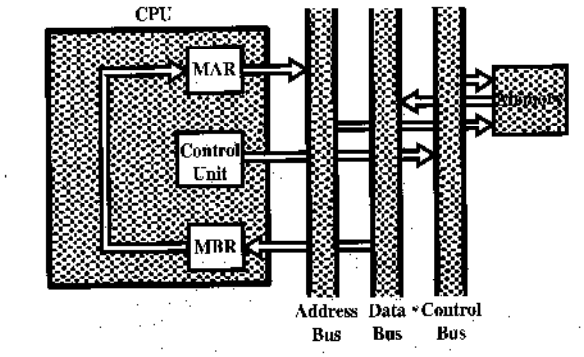
The registers used above, besides the ones described earlier, are the Memory Address Register (MAR) and the Memory Data Register (MDR), which are used (at least conceptually) in the accessing of memory. Often, the MDR is expressed as the MBR (Memory Buffer Register).

We need to make several comments about the above sequence of microoperations. We assume that a clock is available for timing purposes and that it emits regularly spaced clock pulses. Each clock pulse defines a time unit. Thus, all time units are of equal duration. Each microoperation can be performed within the time of a single time unit. The notation (t_1 , t_2 , t_3) represents successive time units. In other words, we have

- **First time unit:** Move contents of PC to MAR.
- **Second time unit**: Move contents of memory location specified by MAR to MBR. Increment by 1 the contents of the PC.
- **Third time unit**: Move contents of MBR to IR.

The Indirect Cycle

Once an instruction is fetched, the next step is to fetch source operands. Continuing our simple example, let us assume a one-address instruction format, with direct and indirect addressing allowed. If the instruction specifies an indirect address, then an indirect cycle must precede the execute cycle.



Data Flow, Indirect Cycle.

The data flow differs somewhat from that indicated in the figure below and includes the following microoperations:

t_1: MAR ← [IR(Address)] //The address field of the instruction is transferred to the MAR.

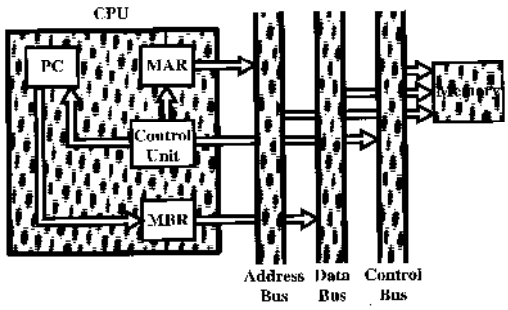
t_2: MBR ← Memory //Fetch the address of the operand from memory and transters it to MBR.

t_3: IR(Address) ← [MBR(Address)] //The address field of the IR is updated from the MBR, so that it now contains a direct rather than an indirect address

The IR is now in the same state as if indirect addressing had not been used, and it is ready for the execute cycle. We skip that cycle for a moment, to consider the interrupt cycle.

The Interrupt Cycle

At the completion of the execute cycle, a test is made to determine whether any enabled interrupts have occurred. The nature of this cycle varies greatly from one machine to another. We present a very simple sequence of events, as illustrated in the figure below.



Data Flow, Interrupt Cycle.

We have:

t_1: MAR ← [PC]
t_2: MAR ← Save_Address
PC ← Routine_Address
t_3: Memory ← [MBR]

In the first step, the contents of the PC are transferred to the MBR., so that they can be saved for return from the interrupt. Then the MAR is loaded with the address at which the contents of the PC are to be saved., and the PC is loaded with the address of the start of the interrupt-processing routine. These two actions may each be a single microoperation. However, because most processors provide multiple types and/or level of interupts, it may take one or more additional microoperations to obtain the save_address and the routine_adress before they can be transferred to the MAR and PC, respectively. In any case, once this is done, the final step is to

store the MBR, which contains the old value of the PC, into memory. The processor is now ready to begin the next instruction cycle.

The Execute Cycle

The fetch, indirect, and interrupt cycles are simple and predictable. Each involves a small, fixed sequence of microoperations and, in each case; the same microoperations are repeated each time around. This is not true for the execute cycle. For a machine with N different opcodes, there are N different sequences of microoperations that can occur. Let us consider several hypothetical examples.

First, consider an addition instruction:

Add R1, X

which adds the contents of the location X to register R1. The following sequence of microoperation might occur:

t_1: MAR ← [IR(address)]
t_2: MBR ← Memory
t_3: R1 ← [R1] + [MBR]

We begin with the IR containing the ADD instruction. In the first step, the address portion of the IR is loaded into the MAR. Then the referenced memory Location is read. Finally, the contents of R1 and MBR are added by the ALU. Again, this is a simplified example. Additional microoperations may be required to extract the register reference from the IR and perhaps to stage the ALU inputs or outputs in some intermediate registers.

Let us look at two more complex examples. A common instruction is increment and skips if zero:

ISZ X

The content of location X is incremented by 1. If the result is 0, the next instruction is skipped. A possible sequence of microinstructions is:

t_1: MAR ← [IR(address)]
t_2: MBR ← Memory
t_3: MBR ← [MBR] + 1
t_4: Memory ← [MBR]
If([MBR] = 0) then (PC ← [PC] + 1)

The new feature introduced here is the conditional action. The PC is incremented if [MBR] = 0. This test and action can be implemented as one microinstruction. Note also that this microinstruction can be performed during the same time unit during which the updated value in MBR is stored back to memory. Finally, consider a subroutine call instruction. As an example, consider a branch-and-save-address instruction:

t_1: MAR ← [IR(address)]
MBR ← [PC]
t_2: PC ← [IR(address)]
Memory ← [MBR]
t_3: PC ← [PC] + 1

The address in the PC at the start of the instruction is the address of the next instruction in sequence. This is saved at the address designated in the IR. The latter address is also incremented to provide the of the instruction for the next instruction cycle.

6. What are the different types of addressing modes? Give an example of each with explanation.

Solution (Hint: see Lecture 21g):

Addressing Modes

To perform any operation, we have to give the corresponding instructions to the microprocessor. In each instruction, programmer has to specify 3 things: Operation to be performed, Address of source of data, Address of destination result. The method by which the address of source of data or the address of destination of result is given in the instruction is called Addressing Modes. The instructions presented here are not valid for all types of microprocessors. If you are using ARM7 all data to be manipulated must be fist moved to registers. In other words, some of the instructions presented here will not be valid for ARM7.

The way by which the microprocessor identifies the operands for a particular instruction is known as **Addressing mode**. An "addressing mode" refers to how you are addressing a given memory location.

To perform any operation, we have to give the corresponding instructions to the microprocessor. In each instruction, programmer has to specify 3 things: Operation to be performed, Address of source of data, Address of destination result. The method by which the address of source of data or the address of destination of result is given in the instruction is called Addressing Modes.

The term addressing mode refers to the way in which the operand of instruction is specified. Intel 8085 uses the following addressing modes:

1. Immediate Addressing Mode
2. Direct Addressing Mode
3. Register Indirect Addressing Mode
4. Register Addressing Mode
5. Indexed Addressing Mode
6. PC-Relative Addressing Mode

1. Immediate Addressing Mode:

One of the simplest modes is immediate addressing, where the operand itself is accessed. Immediate addressing is so-named because the value of constant to be stored in memory immediately follows the operation code in memory. That is to say, the instruction itself dictates what value will be stored in memory. In this mode, the operand is specified within the instruction itself. In other words, the operand specified is not address; it is the actual data to be used.

LDR R1, #1999 R1 ← 1999

a. **LDR is the operation**
b. **1999 is the immediate data (source)**
c. **R1 is the destination**

In this example, the data value 1999 is moved to R1. This instruction is not valid for ARM7. For ARM7, we should instead perform one of these 2 instructions: **MOV R1, #1999** or **LDR R1, =1999**.

Here is another example:

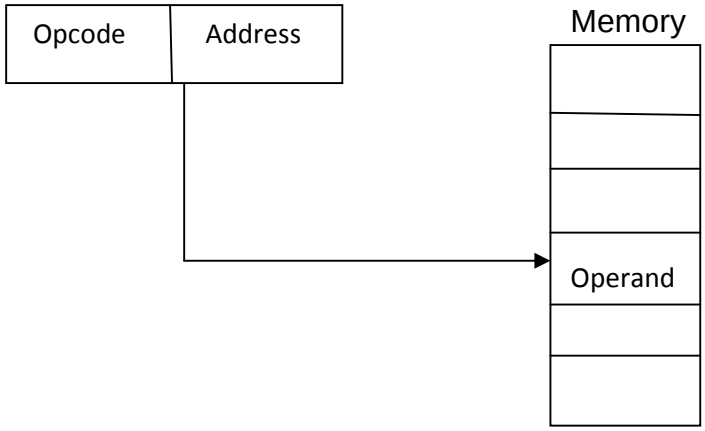
MOV R1, #20H R1 ← 20

This instruction uses Immediate Addressing because the value that immediately follows the register R1 will be moved to R1; in this case 20 (hexadecimal).

Immediate addressing is very fast since the value to be loaded is included in the instruction. However, since the value to be loaded is fixed at compile-time it is not very flexible.

2. Direct Addressing Mode

Another possible mode is direct addressing, where the operand is a constant that represents a memory address. Direct addressing is so-named because the value to be stored in memory is obtained by directly retrieving it from another memory location. In other words, the instruction in this mode includes a memory address; the CPU accesses that location in memory.



LDR R1, 500 R1 ← M[500]

a. **LDR is the operation**
b. **500 is the effective address of source**
c. **R1 is the destination**

- Here the effective address is 500, the same as the operand.
- This is useful for working with pointers.
  – You can think of the constant as a pointer.
  – The register gets loaded with the data at that address.

In the above example, the instruction LDR R1, 500 reads the data from memory location 500 and stores the data in the register R1. This mode is typically used to load operands and values of variables into the CPU.

Again the above instruction will not be valid for ARM7. In ARM7, the memory address must be first in a given register and thereafter this address should be transferred into R1. In other words, the above instruction should be replaced by:

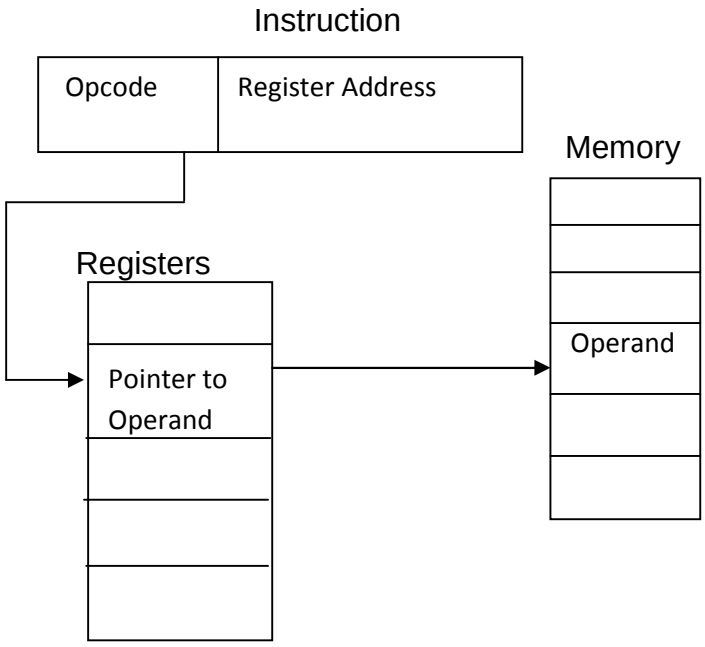
```
MOV  R0, #500
LDR  R1, [R0]
```

Direct addressing is generally fast since, although the value to be loaded isn't included in the instruction, it is quickly accessible since it is stored in the 8051s Internal RAM. It is also much more flexible than Immediate Addressing since the value to be loaded is whatever is found at the given address--which may be variable.

3. Register Indirect Addressing Mode:

We already saw register indirect mode, where the operand is a register that contains a memory address. Indirect addressing is a very powerful addressing mode which in many cases provides an exceptional level of flexibility. Indirect addressing is also the only way to access the extra 128 bytes of Internal RAM found on an 8052 processor.

This mode is similar to indirect addressing. The address field of the instruction refers to a register. The register contains the effective address of the operand. This mode uses one memory reference to obtain the operand. The address space is limited to the width of the registers available to store the effective address.



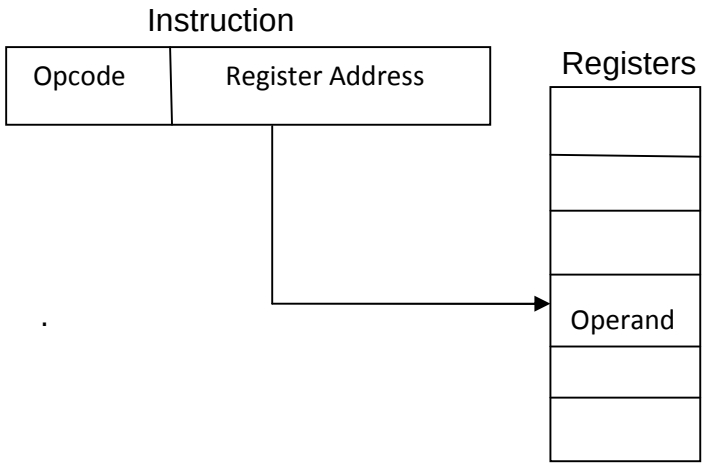
LDR R1, [R0] R1 ← M[R0]

a. **LDR is the operation**
b. **R0 contain the effective address of the source of data**
c. **R1 is the destination**

The above instruction moves the contents of data pointed to by R0 to R1. This achieves R1 = *(R0). This addressing mode is using a register for a base address

4. Register Addressing Mode:

In this mode, the operand is in general purpose register. Register addressing mode is similar to direct addressing. The only difference is that the address field of the instruction refers to a register rather than a memory location 3 or 4 bits are used as address field to reference 8 to 16 generate purpose registers. The advantages of register addressing are Small address field is needed in the instruction.



.

MOV R1, R0 R1 ← R0

a. **MOV is the operation**
b. **R0 is the source data**
c. **R1 is the destination**

Here is another example

ADD R1, R0 R1 ← R1 + R0

a. **ADD is the operation**
b. **R0+R1 are the source data**
c. **R1 is the destination**

5. Indexed Addressing Mode

Operands with indexed addressing include a constant and a register. **Indexed addressing (indexing)** is a method of generating an effective address that modifies the specified address given in the instruction by the contents of a specified index register. The modification is usually that of addition of the contents of the index register to the specified address. The automatic modification of index-register contents results in an orderly progression of effective addresses being generated on successive executions of the instruction containing the reference to the index

register. This progression is terminated when the index register reaches a value that has been specified in an index-register handling instruction.

LDR R1, 500[R0] R1 ← M[R0 + 500]

- The effective address is the register data plus the constant. For instance, if R0 = 25, the effective address here would be 525.
- We can use this addressing mode to access arrays also.
  – The constant is the array address, while the register contains an index into the array.
  – The example instruction above might be used to load the 25th element of an array that starts at memory location 500.
- It's possible to use negative constants too, which would let you index arrays backwards.

The above instruction is not valid for ARM7. Instead we could write

LDR R1, [R0, #500]

6. **PC-Relative addressing Mode**

We've seen PC-relative addressing already. The operand is a constant that is added to the program counter to produce the effective memory address.

200: LDR R1, $30 R1 ← M[201 + 30] //NOT valid in ARM7

- The PC usually points to the address of the next instruction, so the effective address here is 231 (assuming the LDR instruction itself uses one word of memory).
- This is similar to indexed addressing, except the PC is used instead of a regular register.
- Relative addressing is often used in jump and branch instructions.
  – For instance, **JMP $30** lets you skip the next 30 instructions.
  – A negative constant lets you jump backwards, which is common in writing loops.

Addressing mode summary (not all are valid for ARM7)

Mode	**Notation**	**Register transfer equivalent**
Immediate	**LDR R1, #CONST**	**R1 ← CONST**
Direct	**LDR R1, CONST**	**R1 ← M[CONST]**
Register indirect	**LDR R1, [R0]**	**R1 ← M[R0]**
Indexed	**LDR R1, CONST[R0]**	**R1 ← M[R0 + CONST]**
Relative	**LDR R1, $CONST**	**R1 ← M[PC + CONST]**
Indirect	**LDR R1, [CONST]**	**R1 ← M[M[CONST]]**

Consider the following instructions:

LDR R1, [R0, #4] R1 ← M[R0 + 4]

This instruction loads the memory contents at address (R0+4) into R1.

LDR R0, [R1, R2] R0 ← M[R1 + R2]

This instruction loads the memory contents at address (R1+R2) into R0.

LDR R0, [R1, R2, LSL#2]

First, the contents of register R2 is left shifted with 2 bits then added to the contents of R1 to form the target R1 to form the target address to retrieve the values from.

LDR R0, [R1], #4

First, the memory contents at address R1 is loaded into R0, then the value of R1 is incremented by #4

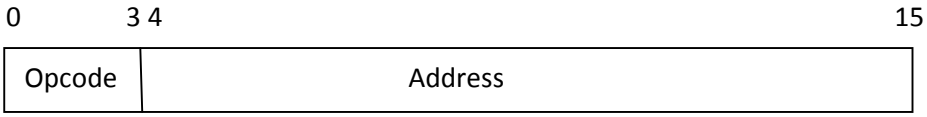
LDR R0, [R1], R2

First, the memory contents at address R1 is loaded into R0, then the value of R1 is incremented by R2

LDR R0, [R1], R2, LSL #2

First, the memory contents at address R1 is loaded into R0, then the value of R1 is updated by adding the value of shifting R2 contents by 2 bits.

7. Consider a 16-bit microprocessor having 16-bit instructions composed of two fields: The first 4 bits contain the opcode and the remainder the immediate operand or an operand address (see Figure below):



a) What is the maximum directly addessable memory capacity?
b) Discuss the impact on the system speed if the microprocessor bus has:
   i. a 16-bit local address bus and a 8-bit local data bus, or
   ii. A 8-bit local address bus and 8-bit local data bus.
c) How many bits are needed for the program counter **PC** and the instruction register **IR**?

Solution (Hint: see Lecture 8 and 14a):

a) If instructions and data are 16 bits long, then it is convenient to organize memory using 16-bit words. As indicated in the figure above, the instruction format provides 4 bits for the opcode, so that there can be as many as $2^4 = 16$ different opcodes, and up to $2^{12} = 4096$ (4K) words of memory can be directly addressed.

b)

i) If the local address bus is 16 bits, the whole address can be transferred at once and decoded in memory. However, since the data bus is only 8 bits, it will require 2 cycles to fetch a 16-bit instruction or operand.

ii) The 8 bits of the address placed on the address bus can not access the whole memory. Thus a more complex memory interface control is needed to latch the first part of the address and then the second part (since the microprocessor will end in two steps).

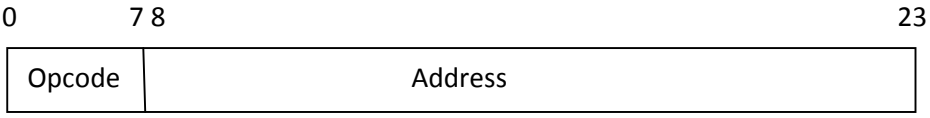
For a 16-bit address, one may assume the first half will decode to access a “row“ in memory, while the second half is sent later to access a “column“ in memory. In addition to the two-step address operation, because of the 8 bits data bus, the microprocessor will need 2 cycles to fetch the 16 bit instruction/operand.

c) The program counter PC must be at least 12 bits. Typically, a 16-bit microprocessor will have a 16bit external address bus and a 16-bit program counter, unless on-chip segment registers are used that may work with a smaller program counter. If the instruction register IR is to contain the whole instruction , it will have to be 16-bits long; if it will contain only the op code, the it will have to be 4 bits long.

8. Consider a hypothetical 24-bit microprocessor having 24-bit instructions composed of two fields: The first byte contains the opcode and the remainder the operand address. What is the maximum directly addressable memory capacity (in bytes)?

Solution (Hint: see Lecture 14a):

Opcode one byte (8 bits) and address 2 Bytes (16 bits), hence



Instruction Format

Therefore, the maximum directly addressable memory capacity (in bytes) → 2^{16} = 64K

9. For the microprocessor in problem 8, discuss the impact on the system speed if the microprocessor bus has a 24-bit local address bus and a 8-bit local data bus.

Solution (Hint: see Lecture 14a):
If the local address bus is 24 bits, the whole address can be transferred at once and decoded in memory. However, since the data bus is only 8 bits, it will require 3 cycles to fetch a 24-bit instruction or operand.

10. For the microprocessor in problem 8, discuss the impact on the system speed if the microprocessor bus has a 12-bit local address bus and a 12-bit local data bus.

Solution (Hint: see Lecture 14a):

The 12 bits of the address placed on the address bus can't access the whole memory. Thus a more complex memory interface control is needed to latch the first part of the address and then the second part (since the microprocessor will end in two steps).

11. For the microprocessor in problem 8, how many bits are needed for the PC and the IR registers?

Solution (Hint: see Lecture 14a):
The program counter must be at least 16 bits. If the instruction register is to contain the whole instruction, it will have to be 24-bit long; if it will contain only the op code (called the op code register) then it will have to be 8 bits long.

12. Consider a hypothetical microprocessor generating a 16-bit address (for example, assume that the program counter and the address registers are 16 bits wide) and having a 16-bit data bus. **(Hint: see Lecture 14a):**

    a. What is the maximum memory address space that the processor can access directly if it is connected to a 16-bit memory?
    b. What is the maximum memory address space that the processor can access directly if it is connected to a 8-bit memory?

Solution (Hint: see Lecture 14a):
In case (a) and (b), the microprocessor will be able to access 2^{16} = 64K bytes; the only difference is that with an 8-bit memory each access will transfer a byte, while with a 16-bit memory an access may transfer a byte or a 16-bit word.

13. Consider a 32-bit microprocessor, with a 16-bit external data bus, driven by an 8 input clock. Assume that this microprocessor has a bus cycle whose minimum duration equal four input clock cycles. What is the maximum data transfer rate that this microprocessor can sustain?

Solution

Clock cycle = 1/input cycle = 1/8 MHz = 125 ns
Bus cycle = 4 x 125 ns = 500 ns
Using a 16-bit external data bus means 2 bytes might be transferred every 500 ns and hence 4 bytes in 1000 ns. It means 4 bytes in 10^3 ns. In other words, $4x10^6$ bytes could be transferred every 10^3 x 10^6 ns. It means the maximum data transfer rate is 4Mbytes/sec. (1 ns = 10^{-9} second)

14. A two segment stage pipeline is used to execute the code segment

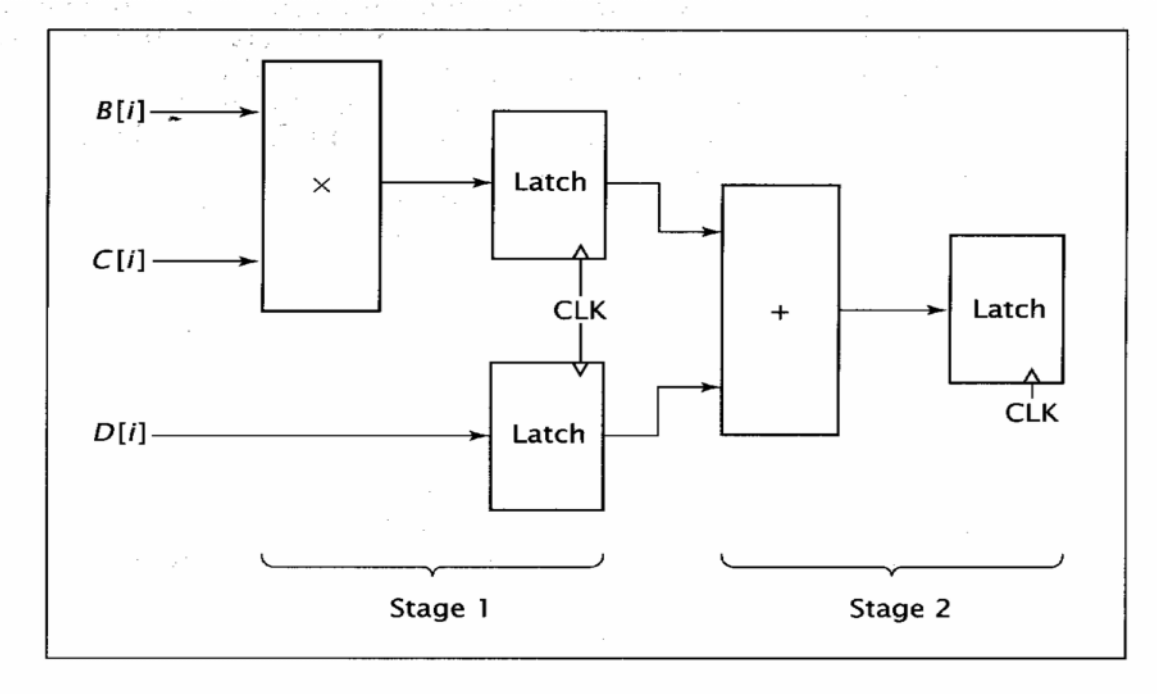
FOR i=1 TO 100 DO {A[i] ← (B[i].C[i]) + D[i]}

Assume that each operation, multiplication and addition, requires 10 ns to complete. During the first 10 ns, the first stage calculates B[1].C[1]. In the next 10 ns, stage 2 adds this value to D[1] and store the result in A[1]. Here we excluded the time needed to fetch the operand and the time delay of the latches to store the result.

a. How much time a non-pipelined uniprocessor takes to execute the code?
b. How much time this 2 segment stage pipeline will take to execute the same code?
c. Determine the speedup ratio if the loop has to be executed 200 times.
d. Determine the speedup ratio if each latch of the k-stage pipeline needs 2 ns to load data.
e. Determine the maximum speedup of a k-stage pipeline.

Solution (see lecture 21a):

a. A no-pipelined uniprocessor takes 20 ns to calculate A[i], and 2,000 ns to execute the entire code.
b. A pipelined unit could break this computation into two stages, as shown in the figure below (see Figure 8.11, page 6 of our lecture note 21a on pipeline).



A two-stage pipeline to implement the program loop

The first stage performs the multiplication and the second performs the addition. During the first 10 ns, the first stage calculates B[1].C[1]. In the next 10 ns, stage 2 adds this value to D[1] and stores the result in A[1]. At the same time, stage 1 multiplies B[2] and C[2]. During the following 10 ns, stage 1 forms B[3].C[3] and stage 2 calculates the final value A[2]. Therefore, in this pipelined processor, the entire code will be executed in 1,010 ns. Compare this with the result obtained in (a).

Remember, the pipeline does not speed up an individual computation. It increases the **throughput**, i.e. the number of results generated per time unit.

c. The speedup S_n to execute n tasks using a k-segment pipeline with a clock cycle time t_p (i.e. speedup of a pipeline processing over an equivalent non-pipeline processing) can be expressed as:

$$S_n = nt_n /[(k+n-1)t_p]$$

Here t_n represents the time taken by a non-pipeline microprocessor to execute each operation (task) of the n tasks, (k+n-1) represents the number of clock cycles required to complete n tasks using a k-segment pipeline, and t_p is the clock time cycle used to execute n tasks on a k-segment pipeline microprocessor. In the above equation, nt_n is the time required to calculate the n tasks on non-pipeline microprocessor and $(k+n-1)t_p$ is the time required to perform the same n tasks using k-stage pipeline.

In our example, t_n= 20 ns and t_p=10 ns. Note that we assumed that the stages of the pipeline have a different clock periods, or stage delays, and t_p is the largest of these periods. Also we defined t_n to be the amount of time needed to process one piece of data using a non-pipelined arithmetic unit. Think of the non-pipelined unit as a one-stage pipeline. The pipelined unit requires k time units, each of duration t_p , to move the first piece of data through the pipeline.

For our example, the speedup can be calculated as follows:

$$S_{200} = 200 \text{ x } 20 \text{ ns} / [(2 + 200 - 1) \text{ x } 10 \text{ ns}] = 1.99$$

The steady-state speedup is the upper limit of the speedup that a pipeline can achieve.

d. In reality, the latches require some amount of time to store their values. This is part of the overhead introduced by pipelines. If each latch needs 2 ns to load data, then each stage requires 12 ns, 10 to calculate its value and 2 to store its result. The pipeline would now require 2,212 ns to calculate A[1…200], which is still faster than the non-pipelined case (4,000 ns). The actual speedup is

$$S_{200} = 200 \text{ x } 20 \text{ ns} / [(2 + 200 - 1) \text{ x } 12 \text{ ns}] = 1.658$$

Note that, if only one value, A[1], is calculated, the speedup is less than one:

$$S_1 = 1 \text{ x } 20 \text{ ns} / [(2 + 1 - 1) \text{ x } 12 \text{ ns}] = 0.833$$

The pipeline is actually slower than the non-pipelined arithmetic unit, due to the delays of the latches at the end of each stage.

e. As $n \rightarrow \infty$, $S_n \rightarrow t_n/t_p$. The maximum speedup occurs when each stage has the same delay. In this case, each of the k stages has a delay of t_n/k, and $t_p=t_n/k$. Thus the maximum speedup of a k-stage pipeline is calculated as follows:

$$S_\infty = t_n/t_p = t_n/(t_n/k) = k = 2$$

If the stages have unequal delays, then some stages will have delays less than t_n/k, while one or more will have a delay greater than t_n/k. Since t_p is the maximum of these stages, this would set $t_p > t_n/k$ and reduce the speedup. For this reason, making the stage delays as close to each other as possible maximizes the speedup of the pipeline.

Exercise

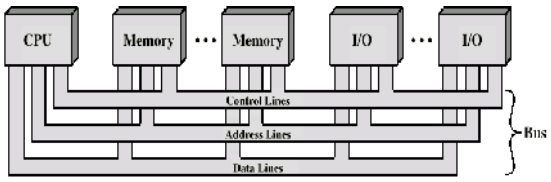
15. In certain scientific computations it is necessary to perform the arithmetic operation $(A_i+B_i)(C_i+D_i)$ with a stream of numbers.
    a. Specify a pipeline configuration to carry out this task (Hint: see figure 8.11, page 6, and lecture 21a).
    b. List the contents of all registers in the pipeline for i=1 through 6

16. Draw a space-time diagram for a six-segment pipeline showing the time it takes to process eight tasks (hint: see the reservation tables on pages 9 and 12 of lecture 21a).

17. Determine the number of clock cycles that it takes to process 200 tasks in a six-segment pipeline.

18. What is a computer bus?

19. What is a bus structure? Describe briefly the function of the lines consisting the bus (Hint: see lecture 14a).



20. Describe the Fetch-Execute Cycle. What are the microoperations (register transfer notations) involved in each cycle? (Hint: see lecture 24).

21. What is bus interrupt? What is interrupt cycle? What are the types of interrupts? (Hint: see lecture 15)

22. Perform the following operations:

    a. Convert the following binary numbers into decimal: 101110 and 110110100.
    b. Convert the following decimal numbers into binary: 2012; 673.
    c. Perform the following binary operations: 100-110000.

Logical conjunction

The truth table for **p AND q** (also written as **p ▫ q)**:

Logical Conjunction		
p	***q***	***p ▫ q***
T	T	T
T	F	F
F	T	F
F	F	F

In ordinary language terms, if both *p* and *q* are true, then the conjunction *p ▫ q* is true. For all other assignments of logical values to *p* and to *q* the conjunction *p ▫ q* is false.
It can also be said that if *p*, then *p ▫ q* is *q*, otherwise *p ▫ q* is *p*.

Logical disjunction

The truth table for **p OR q** (also written as **p ▫ q)**

Logical Disjunction		
p	***q***	***p ▫ q***
T	T	T
T	F	T
F	T	T
F	F	F

Logical disjunction is an operation on two logical values, typically the values of two propositions, that produces a value of *true* if at least one of its operands is true.

23. Let **p** and **q** are two binary numbers such that: **p** = 01100110 and **q** = 01001011. Perform each of the following operations: **p + q**; **p - q**; **q - p**; **p ▫ q**; **p ▫ q**.

24. Use the ARM7 to run the following assembly program. Explain in details the effect of each instruction. Rewrite the program

```
AREA nn, DATA, READWRITE
        num1  DCB 8
        num2  DCB 9
AREA code, CODE, READWRITE
ENTRY
        LDR R1, =num1       ;what is the contents of R1?
        LDR R2, =num2       ;what is the contents of R2?
        LDRB R3, [R2]       ;what is the contents of R3?
        ADD R4, R0, R3      ;what is the contents of R4?
        SUB R5, R0, R3      ;what is the contents of R5?
        MUL R6, R0, R3      ;what is the contents of R6?
        STR  R6, [R1]       ;what is the contents of R6 now?
END
```