Advanced Computer Architecture

(Assignment –I)

Submitted in partial fulfilment of the requirements for the degree of

Master of Technology in Information Technology

by

Vijayananda D Mohire

(Enrolment No.921DMTE0113)

Information Technology Department

Karnataka State Open University

Manasagangotri, Mysore – 570006

Karnataka, India

(2009)

Advanced Computer Architecture

CERTIFICATE

This is to certify that the Assignment-I entitled (Advanced Computer Architecture, subject code: MT12) submitted by Vijayananda D Mohire having Roll Number 921DMTE0113 for the partial fulfilment of the requirements of Master of Technology in Information Technology degree of Karnataka State Open University, Mysore, embodies the bonafide work done by him under my supervision.

Place: _____ **Signature of the Internal Supervisor**

Name

Date: _____ **Designation**

For Evaluation

Question Number	Maximum Marks	Marks awarded	Comments, if any
1	1		
2	1		
3	1		
4	1		
5	1		
6	1		
7	1		
8	1		
9	1		
10	1		
TOTAL	10		

Evaluator's Name and Signature Date

Preface

This document has been prepared specially for the assignments of M.Tech – IT I Semester. This is mainly intended for evaluation of assignment of the academic M.Tech - IT, I semester. I have made a sincere attempt to gather and study the best answers to the assignment questions and have attempted the responses to the questions. I am confident that the evaluator's will find this submission informative and evaluate based on the provide content.

For clarity and ease of use there is a Table of contents and Evaluators section to make easier navigation and recording of the marks. A list of references has been provided in the last page – Bibliography that provides the source of information both internal and external. Evaluator's are welcome to provide the necessary comments against each response, suitable space has been provided at the end of each response.

I am grateful to the Infysys academy, Koramangala, Bangalore in making this a big success. Many thanks for the timely help and attention in making this possible within specified timeframe. Special thanks to Mr. Vivek and Mr. Prakash for their timely help and guidance.

Candidate's Name and Signature Date

Table of Contents

Table of Figures

ADVANCED COMPUTER ARCHITECTURE

RESPONSE TO ASSIGNMENT – I

Question 1 What are the difference types of addressing modes?

Answer 1

Address Modes (Lobur, 2003)

Immediate Addressing

Immediate addressing is so-named because the value to be referenced immediately follows the operation code in the instruction. That is to say, the data to be operated on is part of the instruction. For example, if the addressing mode of the operand is immediate and the instruction is Load 008, the numeric value 8 is loaded into the AC. The 12 bits of the operand field do not specify an address-they specify the actual operand the instruction requires. Immediate addressing is very fast because the value to be loaded is included in the instruction. However, because the value to be loaded is fixed at compile time it is not very flexible.

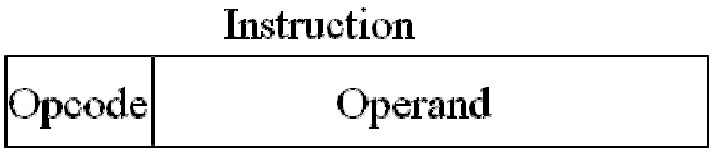


Figure 1 Immediate Addressing (Lovegren, 2007)

Direct Addressing

Direct addressing is so-named because the value to be referenced is obtained by specifying its memory address directly in the instruction. For example, if the addressing mode of the operand is direct and the instruction is Load 008, the data value found at memory address 008 is loaded into the AC. Direct addressing is typically quite fast because, although the value to be loaded is not included in the instruction, it is quickly accessible. It is also much more flexible than immediate addressing because the value to be loaded is whatever is found at the given address, which may be variable.

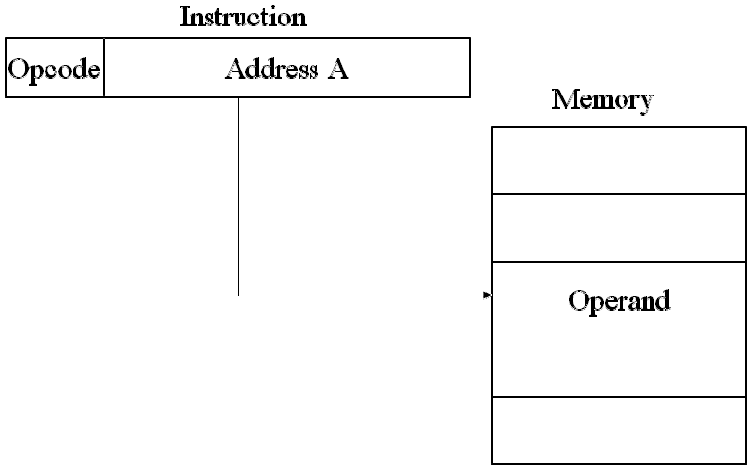


Figure 2 Direct Addressing (Lovegren, 2007)

Register Addressing

In register addressing, a register, instead of memory, is used to specify the operand. This is very similar to direct addressing, except that instead of a memory address, the address field contains a register reference. The contents of that register are

used as the operand.

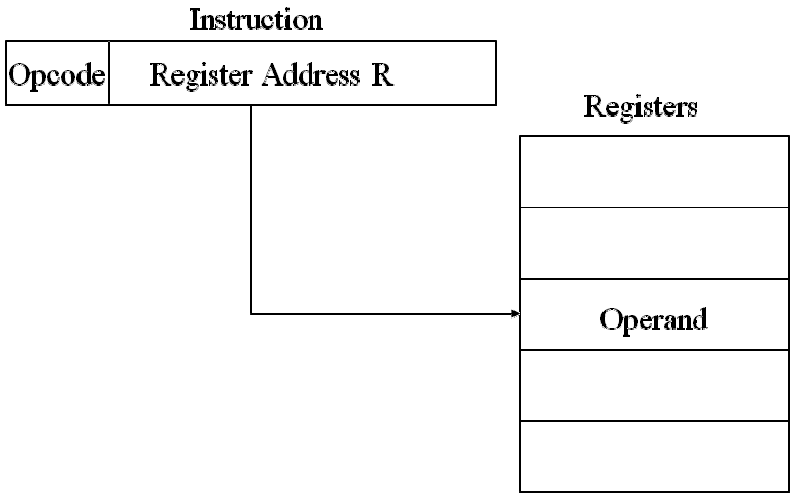


Figure 3 Register Addressing (Lovegren, 2007)

Indirect Addressing

Indirect addressing is a very powerful addressing mode that provides an exceptional level of flexibility. In this mode, the bits in the address field specify a memory address that is to be used as a pointer. The effective address of the operand is found by going to this memory address. For example, if the addressing mode of the operand is indirect and the instruction is Load 008, the data value found at memory address 008 is actually the effective address of the desired operand. Suppose we find the value 2A0 stored in location 008. 2A0 is the "real" address of the value we want. The value found at location 2A0 is then loaded into the AC.

In a variation on this scheme, the operand bits specify a register instead of a memory address. This mode, known as register indirect addressing, works exactly the same way as indirect addressing mode, except it uses a register instead of a

memory address to point to the data. For example, if the instruction is Load R1 and we are using register indirect addressing mode, we would find the effective address of the desired operand in R1.

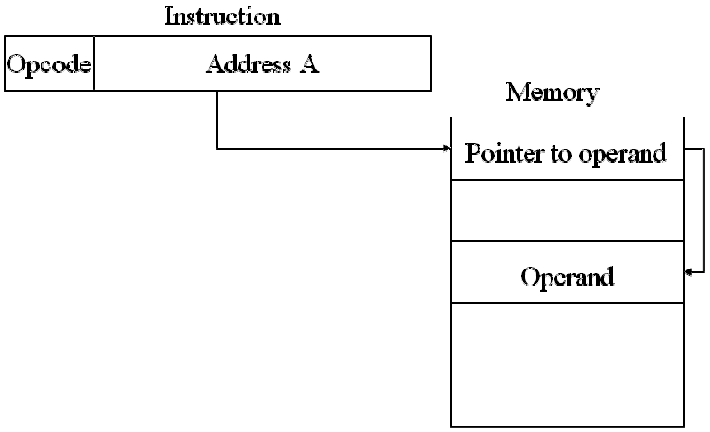


Figure 4 Indirect Addressing (Lovegren, 2007)

Indexed and Based Addressing

In indexed addressing mode, an index register (either explicitly or implicitly designated) is used to store an offset (or displacement), which is added to the operand, resulting in the effective address of the data. For example, if the operand X of the instruction Load X is to be addressed using indexed addressing, assuming R1 is the index register and holds the value 1, the effective address of the operand is actually X + 1. Based addressing mode is similar, except a base address register, rather than an index register, is used. In theory, the difference between these two

modes is in how they are used, not how the operands are computed. An index register holds an index that is used as an offset, relative to the address given in the address field of the instruction. A base register holds a base address, where the address field represents a displacement from this base. These two addressing modes are quite useful for accessing array elements as well as characters in strings. In fact, most assembly languages provide special index registers that are implied in many string operations. Depending on the instruction-set design, general-purpose registers may also be used in this mode.

Stack Addressing

If stack addressing mode is used, the operand is assumed to be on the stack

Evaluator's Comments if any:

Question 2 Write the difference b/w RISC & CISC?

Answer 2

CISC machines rely on microcode to tackle instruction complexity. Microcode tells the processor how to execute each instruction. For performance reasons, microcode is compact, efficient, and it certainly must be correct. Microcode efficiency, however, is limited by variable length instructions, which slow the decoding process, and a varying number of clock cycles per instruction, which makes it difficult to implement instruction pipelines. Moreover, microcode interprets each instruction as it is fetched from memory. This additional translation process takes time. The more complex the instruction set, the more time it takes to look up the instruction and engage the hardware suitable for its execution. (Null, 2003)

RISC architectures take a different approach. Most RISC instructions execute in one clock cycle. To accomplish this speedup, microprogrammed control is replaced by hardwired control, which is faster at executing instructions. This makes it easier to do instruction pipelining, but more difficult to deal with complexity at the hardware level. In RISC systems, the complexity removed from the instruction set is pushed up a level into the domain of the compiler.

The fact that RISC clock cycles are often shorter than CISC clock cycles, and it should be clear that even though there are more instructions, the actual execution time is less for RISC than for CISC. This is the main inspiration behind the RISC design.

The Characteristic differences of RISC Machines versus CISC Machines

RISC	CISC
Multiple register sets, often consisting of more than 256 registers	Single register set, typically 6 to 16 registers total
Three register operands allowed per instruction (e.g., add R1, R2, R3)	One or two register operands allowed per instruction (e.g., add R1, R2)
Parameter passing through efficient on-chip register windows	Parameter passing through inefficient off-chip memory
Single-cycle instructions (except for load and store)	Multiple-cycle instructions
Hardwired control	Micro-programmed control
Highly pipelined	Less pipelined
Simple instructions that are few in number	Many complex instructions
Fixed length instructions	Variable length instructions

Complexity in compiler	Complexity in microcode
Only load and store instructions can access memory	Many instructions can access memory
Few addressing modes	Many addressing modes

Table 1 RISC Vs CISC

The difference between CISC and RISC becomes evident through the basic computer performance equation:

$$\text{CPU Time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{avg. cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

RISC systems shorten execution time by reducing the ***clock cycles per instruction*** (i.e. simple instructions take less time to interpret)

CISC systems shorten execution time by reducing the ***number of instructions per program***

Evaluator's Comments if any:

Question 3 What is the principle of Virtual memory?

Answer 3

The purpose of virtual memory is to use the hard disk as an extension of RAM, thus increasing the available address space a process can use. Most personal computers have a relatively small amount (typically less than 512MB) of main memory. This is usually not enough memory to hold multiple applications concurrently, such as a word processing application, an e-mail program, and a graphics program, in addition to the operating system itself. Using virtual memory, your computer addresses more main memory than it actually has, and it uses the hard drive to hold the excess. This area on the hard drive is called a page file, because it holds chunks of main memory on the hard drive. The easiest way to think about virtual memory is to conceptualize it as an imaginary memory location in which all addressing issues are handled by the operating system. (Null L. , 2003)

The most common way to implement virtual memory is by using paging, a method in which main memory is divided into fixed-size blocks and programs are divided into the same size blocks. Typically, chunks of the program are brought into memory as needed. It is not necessary to store contiguous chunks of the program in contiguous

chunks of main memory. Because pieces of the program can be stored out of order, program addresses, once generated by the CPU, must be translated to main memory addresses. Remember, in caching, a main memory address had to be transformed into a cache location. The same is true when using virtual memory; every virtual address must be translated into a physical address. How is this done? Before delving further into an explanation of virtual memory, let's define some frequently used terms for virtual memory implemented through paging:

- *Virtual address*-The logical or program address that the process uses. Whenever the CPU generates an address, it is always in terms of virtual address space.
- *Physical address*-The real address in physical memory.
- *Mapping*-The mechanism by which virtual addresses are translated into physical ones (very similar to cache mapping)
- *Page frames*-The equal-size chunks or blocks into which main memory (physical memory) is divided.
- *Pages*-The chunks or blocks into which virtual memory (the logical address space) is divided, each equal in size to a page frame. Virtual pages are stored on disk until needed.
- *Paging*-The process of copying a virtual page from disk to a page frame in main memory.

- *Fragmentation*-Memory that becomes unusable.
- *Page fault*-An event that occurs when a requested page is not in main memory and must be copied into memory from disk.

Because main memory and virtual memory are divided into equal size pages, pieces of the process address space can be moved into main memory but need not be stored contiguously. As previously stated, we need not have all of the process in main memory at once; virtual memory allows a program to run when only specific pieces are present in memory. The parts not currently being used are stored in the page file on disk.

Virtual memory can be implemented with different techniques, including paging, segmentation, or a combination of both, but paging is the most popular. (This topic is covered in great detail within the study of operating systems.) The success of paging, like that of cache, is very dependent on the locality principle. When data is needed that does not reside in main memory, the entire block in which it resides is copied from disk to main memory, in hopes that other data on the same page will be useful as the program continues to execute.

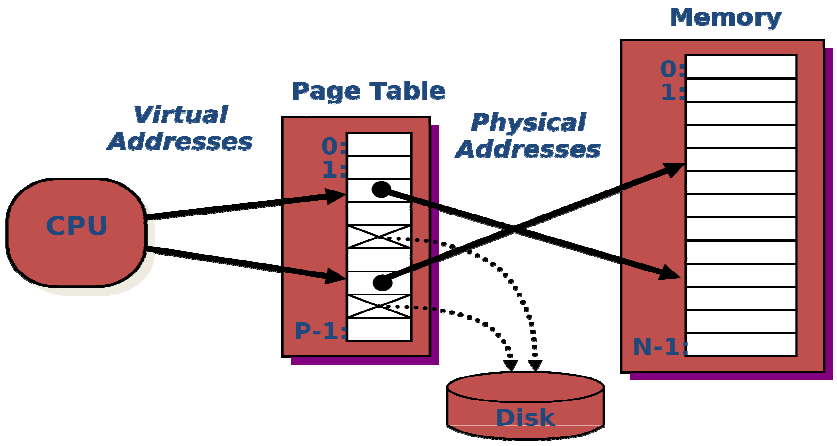


Figure 5 Virtual Address (O'Hallaron, 2003)

Address Translation: Hardware converts *virtual addresses* to *physical addresses* via an OS-managed lookup table (*page table*)

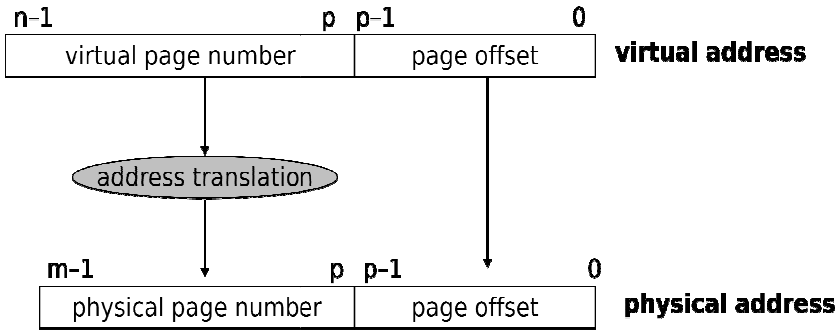


Figure 6 Virtual address translation (O'Hallaron, 2003)

Parameters

- **P = 2^p = page size (bytes).**
- **N = 2^n = Virtual address limit**
- **M = 2^m = Physical address limit**

Evaluator's Comments if any:

Question 4 Why Cache is needed? Explain?

Answer 4

Cache is used by the central processing unit of a computer to reduce the average time to access memory. The cache is a smaller, faster memory which stores copies of the data from the most frequently used main memory locations. As long as most memory accesses are cached memory locations, the average latency of memory accesses will be closer to the cache latency than to the latency of main memory. (Null L. , Cache memory, 2003)

When the processor needs to read from or write to a location in main memory, it first

checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory.

The diagram below shows two memories. Each location in each memory has a datum (a cache line), which in different designs ranges in size from 8 to 512 bytes. The size of the cache line is usually larger than the size of the usual access requested by a CPU instruction, which ranges from 1 to 16 bytes. Each location in each memory also has an index, which is a unique number used to refer to that location. The index for a location in main memory is called an address. Each location in the cache has a tag that contains the index of the datum in main memory that has been cached. In a CPU's data cache these entries are called cache lines or cache blocks.

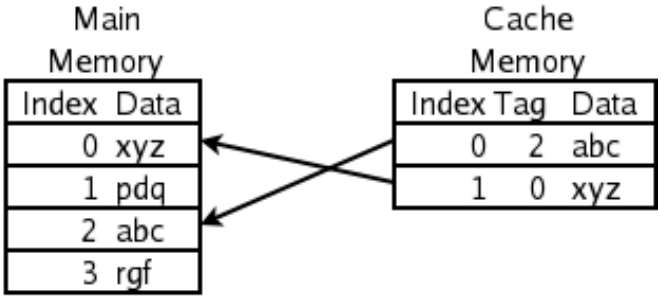


Figure 7 Cache Memory (Anonymous, 2009)

The purpose of cache is to speed up memory accesses by storing recently used

data closer to the CPU, instead of storing it in main memory. Although cache is not as large as main memory, it is considerably faster. Whereas main memory is typically composed of DRAM with, say, a 60ns access time, cache is typically composed of SRAM, providing faster access with a much shorter cycle time than DRAM (a typical cache access time is 10ns). Cache does not need to be very large to perform well. A general rule of thumb is to make cache small enough so that the overall average cost per bit is close to that of main memory, but large enough to be beneficial. Because this fast memory is quite expensive, it is not feasible to use the technology found in cache memory to build all of main memory.

What makes cache "special"? Cache is not accessed by address; it is accessed by content. For this reason, cache is sometimes called content addressable memory or CAM. Under most cache mapping schemes, the cache entries must be checked or searched to see if the value being requested is stored in cache. To simplify this process of locating the desired data, various cache mapping algorithms are used

Evaluator’s Comments if any:

Question 5 What is DMA? How it is used?

Answer 5

Computer systems employ any of four general I/O control methods. These methods are programmed I/O, interrupt-driven I/O, direct memory access, and channel-attached I/O. Although one method isn't necessarily better than another, the manner in which a computer controls its I/O greatly influences overall system design and performance. (Null L. , I/O Architectures, 2003)

Direct Memory Access

With both programmed I/O and interrupt-driven I/O, the CPU moves data to and from the I/O device. During I/O, the CPU runs instructions similar to the following pseudocode:

```
ADD 1 TO Byte-count
IF Byte-count > Total-bytes-to-be-transferred THEN
  EXIT
ENDIF
Place byte in destination buffer
```

Raise byte-ready signal

Initialize timer

REPEAT

WAIT

UNTIL Byte-acknowledged, Timeout, OR Error

ENDWHILE

Clearly, these instructions are simple enough to be programmed in a dedicated chip. This is the idea behind direct memory access (DMA). When a system uses DMA, the CPU offloads execution of tedious I/O instructions. To effect the transfer, the CPU provides the DMA controller with the location of the bytes to be transferred, the number of bytes to be transferred, and the destination device or memory address. This communication usually takes place through special I/O registers on the CPU. A sample DMA configuration is shown in Figure .Once the proper values are placed in memory, the CPU signals the DMA subsystem and proceeds with its next task, while the DMA takes care of the details of the I/O. After the I/O is complete (or ends in error), the DMA subsystem signals the CPU by sending it another interrupt.

As you can see by Figure, the DMA controller and the CPU share the memory bus. Only one of them at a time can have control of the bus, that is, be the bus master. Generally, I/O takes priority over CPU memory fetches for program instructions and

data because many I/O devices operate within tight timing parameters. If they detect no activity within a specified period, they timeout and abort the I/O process. To avoid device timeouts, the DMA uses memory cycles that would otherwise be used by the CPU. This is called cycle stealing. Fortunately, I/O tends to create bursty traffic on the bus: data is sent in blocks, or clusters. The CPU should be granted access to the bus between bursts, though this access may not be of long enough duration to spare the system from accusations of "crawling during I/O."

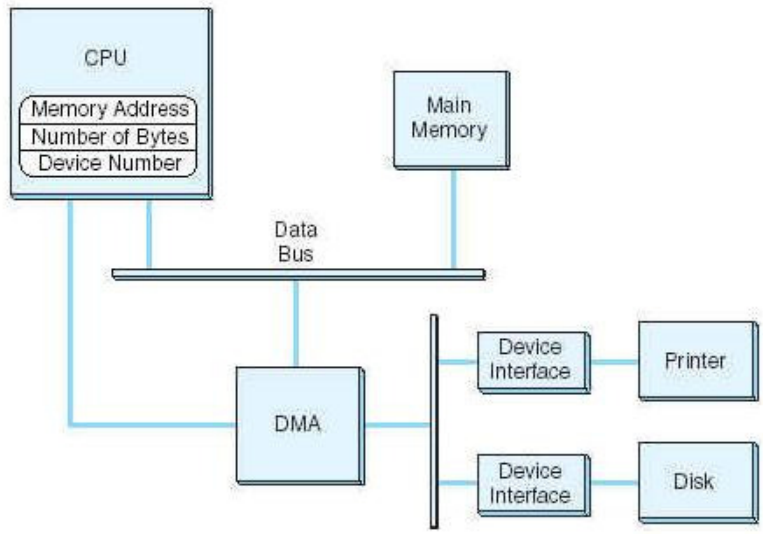


Figure 8 DMA (Null L. , I/O Architectures, 2003)

Evaluator's Comments if any:

Question 6 What are the functions of typical pipelining?

Answer 6

Pipelining is analogous to an automobile assembly line. Each step in a computer pipeline completes a part of an instruction. Like the automobile assembly line, different steps are completing different parts of different instructions in parallel. Each of the steps is called a pipeline stage. The stages are connected to form a pipe. Instructions enter at one end, progress through the various stages, and exit at the other end. The main function is to balance the time taken by each pipeline stage (i.e., more or less the same as the time taken by any other pipeline stage). If the stages are not balanced in time, after awhile, faster stages will be waiting on slower ones

Typical pipeline stages:

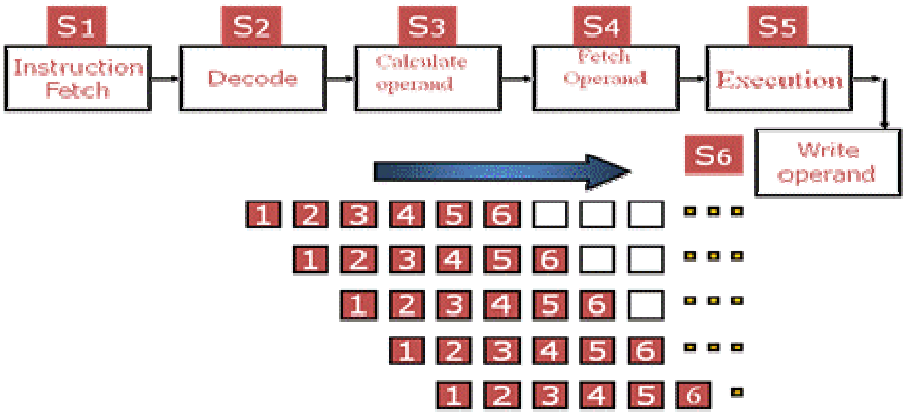


Figure 9 Pipeline Stages

Typical pipeline functions

To provide the above feature the pipeline needs to adhere to certain standards like:

- First, the architecture supports fetching instructions and data in parallel.
- Second, the pipeline can be kept filled at all times. This is not always the case. Pipeline *hazards* arise that cause pipeline conflicts and stalls.
- An instruction pipeline may stall, or be flushed for any of the following reasons:
    1. Resource conflicts.
    2. Data dependencies.
    3. Conditional branching.
- Measures should be taken at the software level as well as at the hardware level to reduce the effects of these hazards.

Evaluator's Comments if any:

Question 7 Explain the different types of dependencies?

Answer 7

- A parallel program is comprised of a number of tasks running as threads (or processes) on a number of processing elements that cooperate/communicate as part of a single parallel computation.

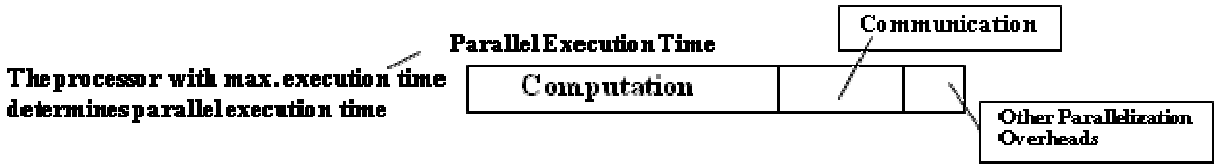


Figure 10 Parallel Execution (Al-Mouhamed, 2009)

- *Task*:
  - Arbitrary piece of under composed work in parallel computation
  - Executed sequentially on a single processor; concurrency in parallel computation is only across tasks, at Thread Level Parallelism (TLP)
- *Parallel or Independent Tasks:*

  - Tasks that with no dependencies among them and thus can run in parallel on different processing elements.

- *Parallel Task Grain Size:* The amount of computations in a task.
- *Process (thread)*:
  - Abstract entity that performs the computations assigned to a task
  - Processes communicate and synchronize to perform their tasks
- *Processor or (Processing Element)*:
  - Physical computing engine on which a process executes sequentially
  - Processes virtualize machine to programmer
    - First write program in terms of processes, then map to processors
- Communication to Computation Ratio (C-to-C Ratio): Represents the amount of resulting communication between tasks of a parallel program. In general, for a parallel computation, a lower C-to-C ratio is desirable and usually indicates better parallel performance. (Al-Mouhamed, 2009)

Dependencies: (Hwang, 2001)

- Program segments cannot be executed in parallel unless they are independent.
- Independence comes in several forms:

- **Data dependence**: data modified by one segment must not be modified by another parallel segment.
- **Control dependence**: if the control flow of segments cannot be identified before run time, then the data dependence between the segments is variable.
- **Resource dependence**: even if several segments are independent in other ways, they cannot be executed in parallel if there aren't sufficient processing resources (e.g. functional units).

Data Dependence: (Hwang, 2001)

- Flow dependence: S1 precedes S2, and at least one output of S1 is input to S2.
- Anti-dependence: S1 precedes S2, and the output of S2 overlaps the input to S1.
- Output dependence: S1 and S2 write to the same output variable.
- I/O dependence: two I/O statements (read/write) reference the same variable, and/or the same file.
- Unknown dependence:
  - The subscript of a variable is itself subscripted.
  - The subscript does not contain the loop index variable.

- A variable appears more than once with subscripts having different coefficients of the loop variable (that is, different functions of the loop variable).
- The subscript is nonlinear in the loop index variable.

- Parallel execution of program segments which do not have total data independence can produce non-deterministic results.

Flow dependence : (Al-Mouhamed, 2009)

- **Assume task S2 follows task S1 in sequential program order**
- **Task S1 produces one or more results used by task S2,**
  - Then task S2 is said to be data dependent on task S1
- **Changing the relative execution order of tasks S1, S2 in the parallel program violates this data dependence and results in incorrect execution.**

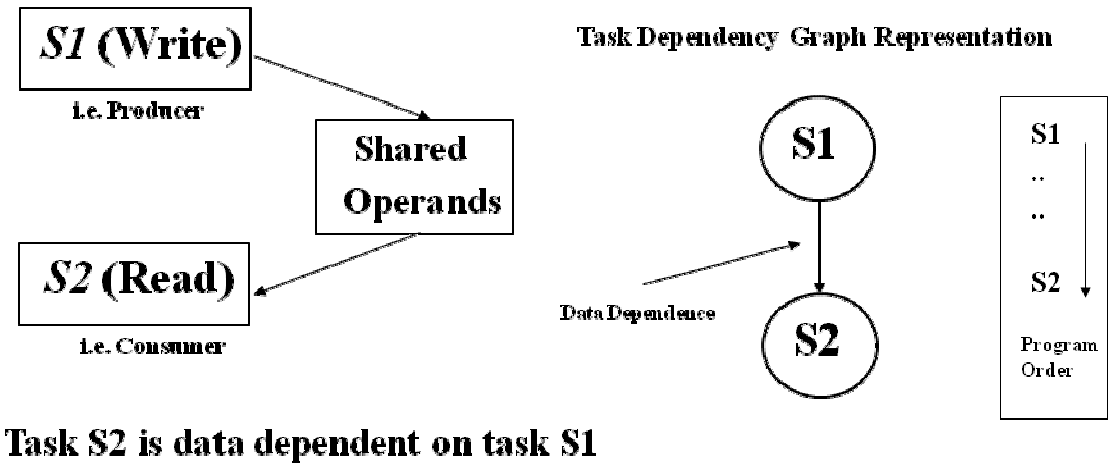


Figure 11 Flow Dependencies (Al-Mouhamed, 2009)

Anti-Dependencies (Al-Mouhamed, 2009)

- **Assume task S2 follows task S1 in sequential program order**
- **Task S1 reads one or more values from one or more names (registers or memory locations)**
- **Instruction S2 writes one or more values to the same names (same registers or memory locations read by S1)**
  - Then task S2 is said to be anti-dependent on task S1
- **Changing the relative execution order of tasks S1, S2 in the parallel program violates this name dependence and may result in incorrect execution.**

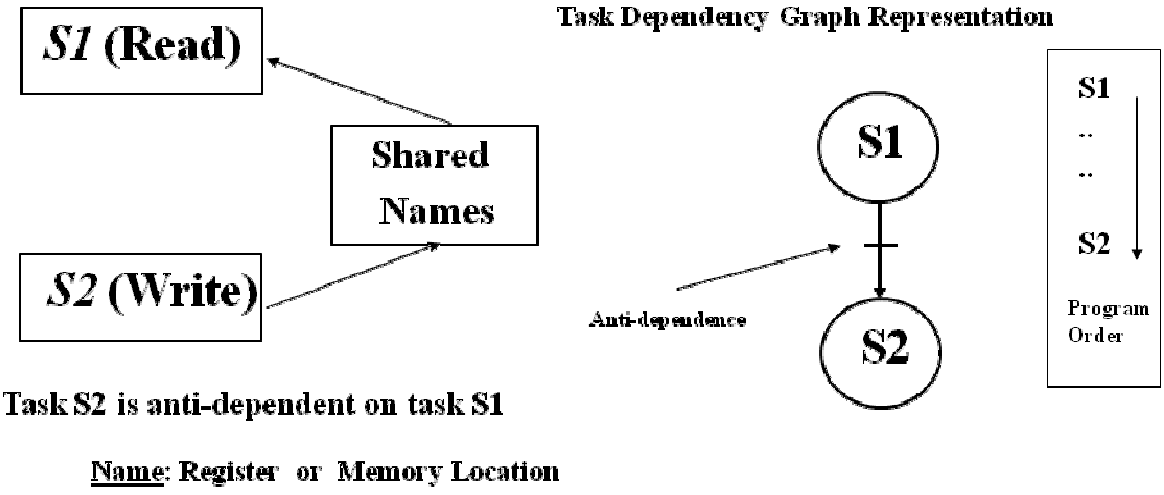


Figure 12 Anti-Dependencies (Al-Mouhamed, 2009)

Output Dependencies (Al-Mouhamed, 2009)

- **Assume task S2 follows task S1 in sequential program order**

- **Both tasks S1, S2 write to the same a name or names (same registers or memory locations)**
  - Then task S2 is said to be output-dependent on task S1
- **Changing the relative execution order of tasks S1, S2 in the parallel program violates this name dependence and may result in incorrect execution.**

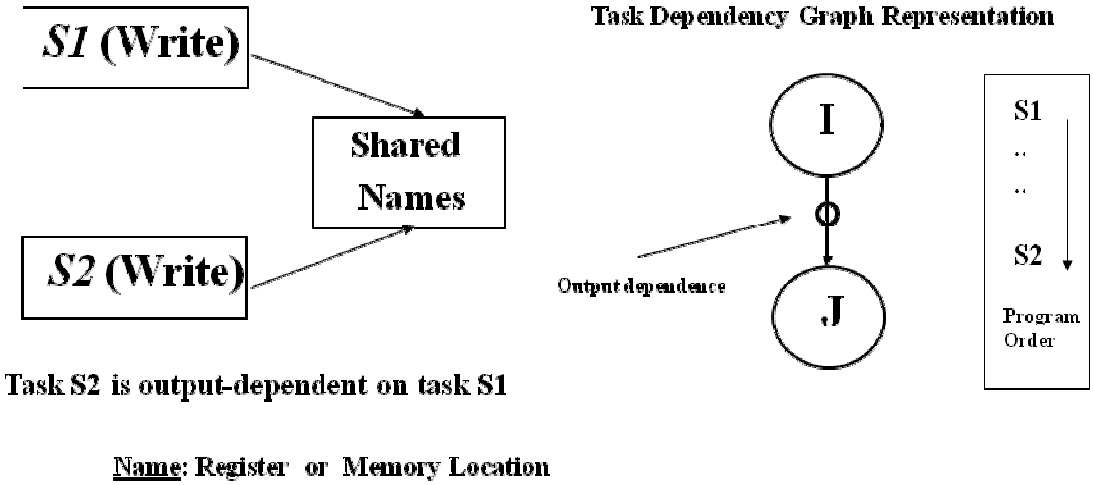


Figure 13 Output Dependencies (Al-Mouhamed, 2009)

Control Dependence:

- *Control-independent example:*

```
for (i=0;i<n;i++) {
        a[i] = c[i];
        if (a[i] < 0) a[i] = 1;
}
```

- *Control-dependent example:*

```
for (i=1;i<n;i++) {
        if (a[i-1] < 0) a[i] = 1;
}
```

- *Compiler techniques are needed to get around control dependence limitations.*

Resource Dependence:

- Data and control dependencies are based on the independence of the work to be done.
- Resource independence is concerned with conflicts in using shared resources, such as registers, integer and floating point ALUs, etc.
- ALU conflicts are called ALU dependence.
- Memory (storage) conflicts are called storage dependence.

Evaluator's Comments if any:

Question 8 Describe Bernstein's condition of parallelism?

Answer 8

- Bernstein's conditions are a set of conditions which must exist if two processes can execute in parallel. (Hwang, 2001)
- Notation
  - I_i is the set of all input variables for a process P_i.
  - O_i is the set of all output variables for a process P_i.
- If P_1 and P_2 can execute in parallel (which is written as $P_1 \parallel P_2$), then:

$$I_1 \cap O_2 = \varnothing$$
$$I_2 \cap O_1 = \varnothing$$
$$O_1 \cap O_2 = \varnothing$$

- In terms of data dependencies, Bernstein's conditions imply that two processes can execute in parallel if they are flow-independent, anti-independent, and output-independent.
- The parallelism relation || is commutative ($P_i \parallel P_j$ implies $P_j \parallel P_i$), but not transitive ($P_i \parallel P_j$ and $P_j \parallel P_k$ does not imply $P_i \parallel P_k$) . Therefore, || is not an equivalence relation.
- Intersection of the input sets is allowed.

Two processes P_1 , P_2 with input sets I_1, I_2 and output sets O_1, O_2 can execute in parallel (denoted by $P_1 \parallel P_2$) if: (Al-Mouhamed, 2009)

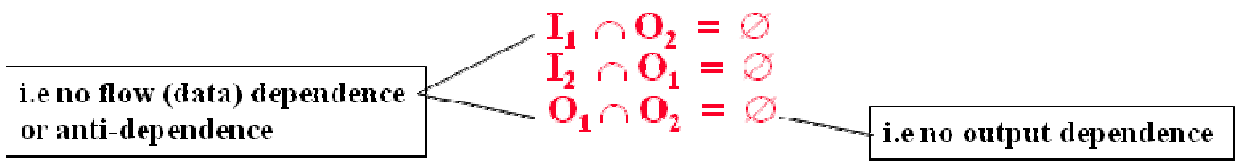


Figure 14 Bernstein parallelism (Al-Mouhamed, 2009)

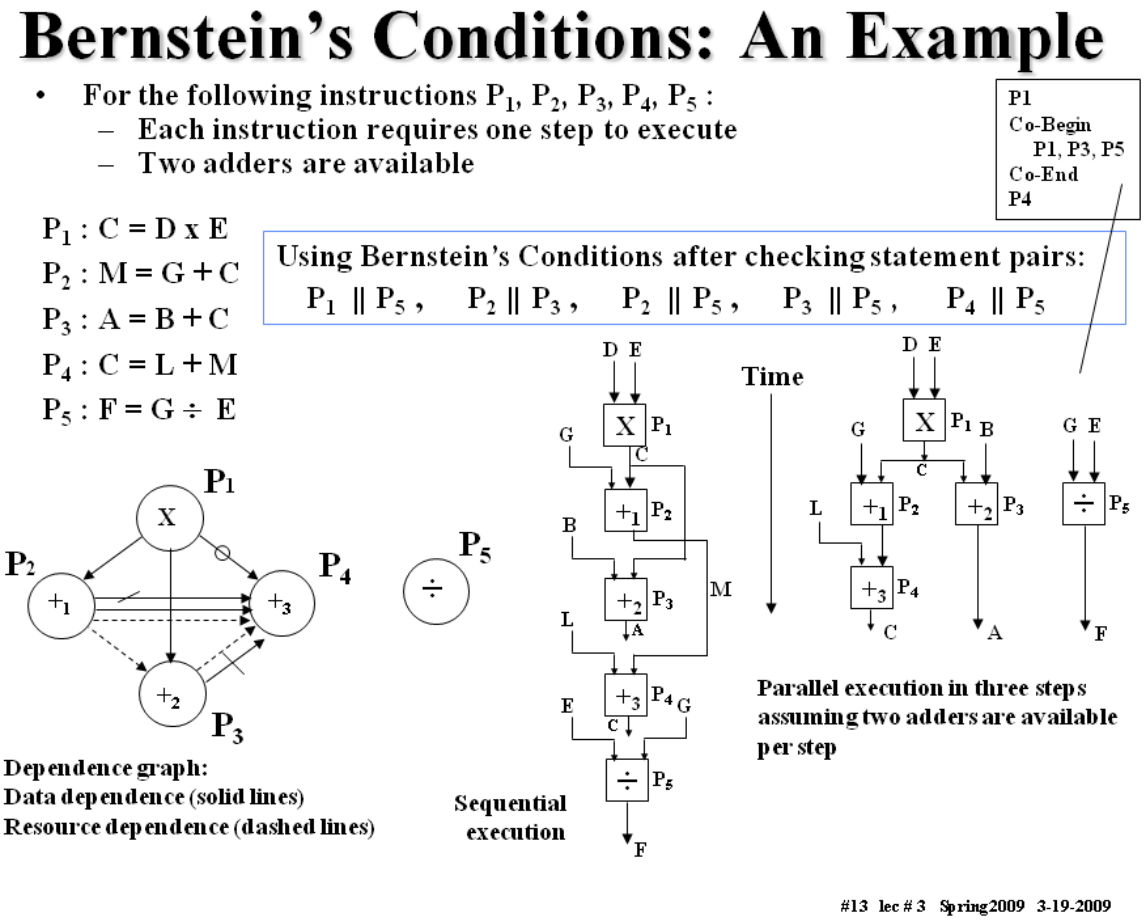


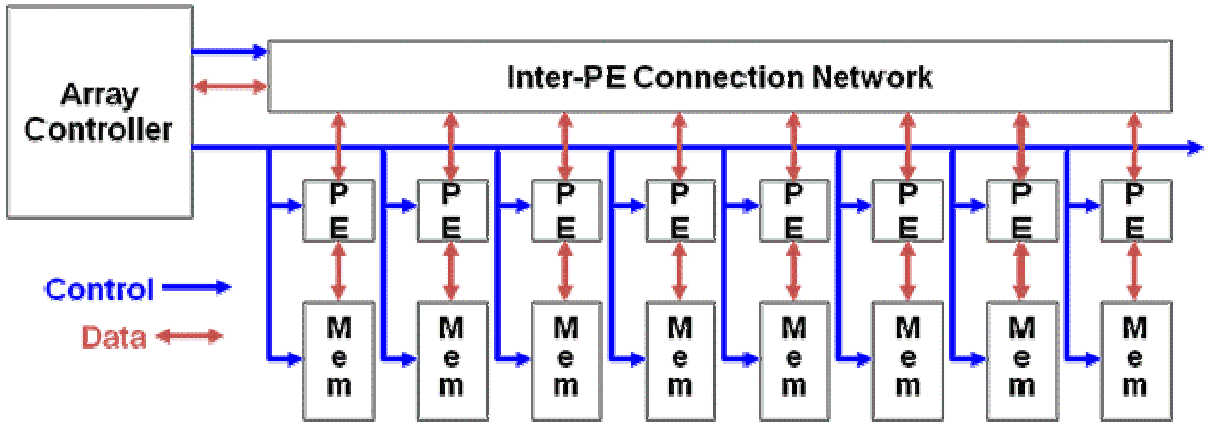
Figure 15 Bernstein Condition: An example (Al-Mouhamed, 2009)

Evaluator's Comments if any:

Question 9 What are the SIMD programming principles?

Answer 9

Before delving into SIMD programming, let's look at the SIMD Architecture as below.



Central controller broadcasts instructions to multiple processing elements (PEs)

- **Only requires one controller for whole array**
- **Only requires storage for one copy of program**
- **All computations fully synchronized**

SIMDs are Array processor type and have features:

- It is composed of N identical processing elements under the control of a single control unit and a number of memory modules.
    - The PEs execute instruction in a lock-step mode.

- Processing units and memory elements communicate with each other through an interconnection network.
    - Different topologies can be used.
- Complexity of the control unit is at the same level of the uniprocessor system.
- Control unit is a computer with its own high speed registers, local memory and arithmetic logic unit.
- The main memory is the aggregate of the memory modules.

Programming principles:

Capable of processing multiple data with a single instruction, SIMD operations are widely used for 3D graphics and audio/video processing in multimedia applications. A number of recently developed processors have instructions for SIMD operations (hereinafter referred to as SIMD instructions). In multimedia extensions for the Intel x86, for example, SSE instructions and MMX instructions are defined as SIMD instructions. The Sony Play station 3 Cell processor described includes two types of SIMD instructions – VMX instructions implemented in the PPE and SPU SIMD instructions implemented in SPEs. (Foundation, 2009)

Here, SIMD programming is explained using the VMX instructions implemented in the PPE. We will look at what SIMD programming is, what data it uses and how it is performed. A number of sample programs are presented to make it easier.

SIMD Operation Overview

Let's start with the unique aspects of SIMD operations. SIMD is primarily geared towards graphics applications and physics calculations that require simple, repetitive calculations of enormous amounts of data.

How SIMD Operates

SIMD is short for Single Instruction/Multiple Data, while the term SIMD operations refers to a computing method that enables processing of multiple data with a single instruction. In contrast, the conventional sequential approach using one instruction to process each individual data is called scalar operations.

Using a simple summation as an example, the difference between the scalar and SIMD operations is illustrated below. See Fig. 16 for how each method handles the same four sets of additions.

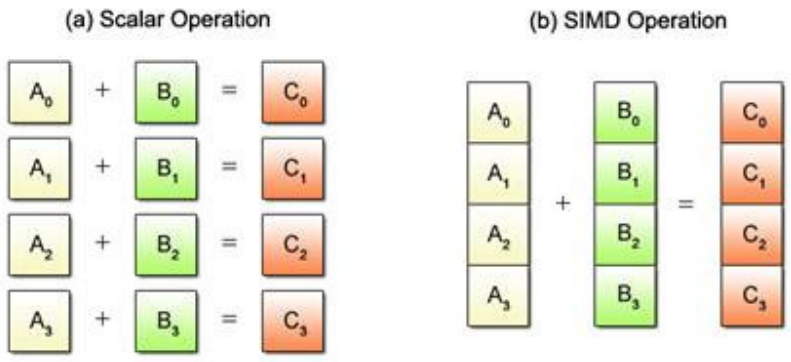


Figure 16 Scalar vs. SIMD Operations (Foundation, 2009)

With conventional scalar operations, four add instructions must be executed one after another to obtain the sums as shown in Fig. 16 (a). Meanwhile, SIMD uses only one add instruction to achieve the same result, as shown in Fig. 16 (b). Requiring fewer instructions to process a given mass of data, SIMD operations yield higher efficiency than scalar operations.

Restrictions on SIMD Operations

Despite the advantage of being able to process multiple data per instruction, SIMD operations can only be applied to certain predefined processing patterns. Fig. 17 shows one such pattern where the same add operation is performed for all data.

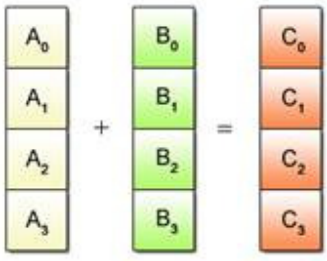


Figure 17 Example of SIMD Processable Patterns

SIMD operations cannot be used to process multiple data in different ways. A typical

example is given in Fig. 18 where some data is to be added and other data is to be deducted, multiplied or divided.

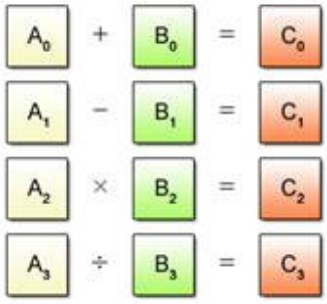


Figure 18 Example of SIMD Unprocesable Patterns

Data Used in SIMD Programming

This section focuses on the data used in SIMD programming.

Vector Type

Conventional data types used in the C programming language, such as *char, int* and *float*, are called scalar types. Data types used for SIMD operations are called vector types. Each vector type has its corresponding scalar type as shown in Table 2

Table 2 List of Vector Types

Vector Type	Data
__*vector unsigned char*	Sixteen unsigned 8-bit data
__*vector signed char*	Sixteen signed 8-bit data
__*vector unsigned short*	Eight unsigned 16-bit data
__*vector signed short*	Eight signed 16-bit data
__*vector unsigned int*	Four unsigned 32-bit data
__*vector signed int*	Four signed 32-bit data
__*vector unsigned long long*	Two unsigned 64-bit data
__*vector signed long long*	Two signed 64-bit data
__*vector float*	Four single-precision floating-point data
__*vector double*	Two double-precision floating-point data

Vector Format (Byte Order*)

The Cell uses 128-bit (16-byte) fixed-length vectors made up of 2 to 16 elements according to the data type. A vector can be interpreted as a series of scalars of the corresponding type (*char, int, float* and so on) in a 16-byte space. As shown in Fig. 19,

byte ordering* and element numbering on the Cell is displayed in big-endian order.

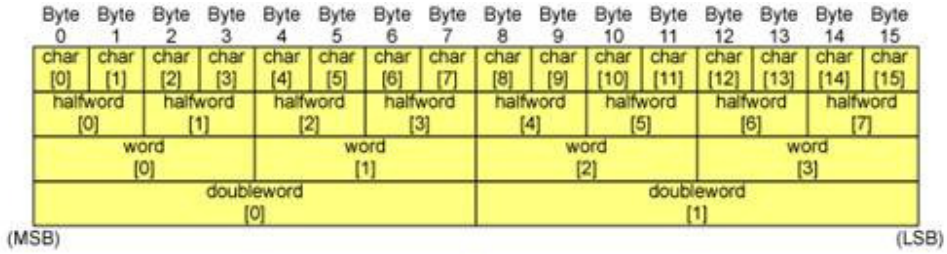


Figure 19 Cell processor Format

Vector Literal

A vector literal is written as a parenthesized vector type followed by a curly braced set of constant expressions. The elements of the vector are initialized to the corresponding expression. Elements for which no expressions are specified default to 0. Vector literals may be used either in initialization statements or as constants in executable statements. Some examples of usage are shown below.

Example (1): Use in variable initialization statement (Vector literal shown in **BOLD**)

```
void func_a(void)
{
  __vector signed int va = (__vector signed int) { -2, -1,  1,  2 };
```

Example (2): Use as a constant in executable statement (Vector literal shown in **BOLD**)

va = vec_add(va, **((__vector signed int) { 1, 2, 3, 4 })**);

When used for macros, the entire vector literal must be enclosed in parentheses.

Table 3 List of vector literals

Notation	Definition
(__ *vector unsigned char*){ *unsigned int*,…}	A set of sixteen unsigned 8-bit data
(__ *vector signed char*){ *signed int*,…}	A set of sixteen signed 8-bit data
(__ *vector unsigned short*){ *unsigned short*,…}	A set of eight unsigned 16-bit data
(__ *vector signed short*){ *signed int*,…}	A set of eight signed 16-bit data
(__ *vector unsigned int*){ *unsigned int*,…}	A set of four unsigned 32-bit data
(__ *vector signed int*){ *signed int*,…}	A set of four signed 32-bit data

(__*vector unsigned long long*){ *unsigned long long*,…}	A set of two unsigned 64-bit data
(__*vector signed long long*){ *signed long long*,…}	A set of two signed 64-bit data
(__*vector float*){ *float*,…}	A set of four 32-bit floating-point data
(__*vector double*){ *double*,…}	A set of two 64-bit floating-point data

Relationship between Vectors and Scalars

With SIMD programming, there often arises a need to refer to a specific vector element as a scalar or to refer to a block of scalars as a single vector. This section describes the referencing method to meet that need, which, for example, makes it possible to output only the third element of a vector or to bundle scalar array input data into vectors suitable for SIMD processing.

Do you remember the vector's data structure illustrated in Fig. 19? Structured in the same manner as a 16-byte array of scalars and allocated in the memory as shown in Fig. 20, a vector (Fig. 20 (a)) can be viewed as a scalar array 16 bytes in data length (Fig. 20 (b)).

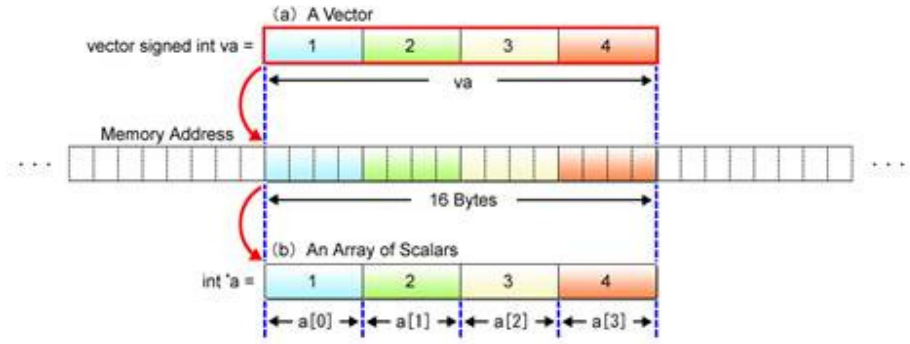


Figure 20 Vector representation

This change in the way of looking at data is equal to pointer casting as performed in C-language programming. A vector can be referenced as a scalar by casting the vector pointer to the scalar pointer. See below for a specific example.

Example (3): In reference to a scalar corresponding to the third element of a vector

(Pointer cast indicated in **BOLD**)

__vector signed int va = (__vector signed int) { 1, 2, 3, 4 };

int *a = (int *) &va;

printf("a[2] = %d¥n", a[2]);

Similarly, a scalar array in the memory can be referenced as vectors. In such cases, the scalar pointer (holding the address of the beginning of the scalar array) is cast to the vector pointer as shown in Example (4).

Example (4): In reference to vectors corresponding to a specific scalar array

(Pointer cast indicated in **BOLD**)

```
int a[8] __attribute__((aligned(16))) = { 1, 2, 3, 4, 5, 6, 7, 8 };
__vector signed int *va = (__vector signed int *) a;
 /* va[0] = { 1, 2, 3, 4}, va[1] = { 5, 6, 7, 8 } */
vb = vec_add(va[0], va[1]);
```

When referring to scalars in an equivalent vector form, the address of the first scalar must be aligned on a 16-byte boundary, i.e., the lower 4 bits of the address must be all "0". If not aligned, you may not be able to refer to vector-form data in a way you expect. A visual representation of byte alignment is given in Fig. 21.

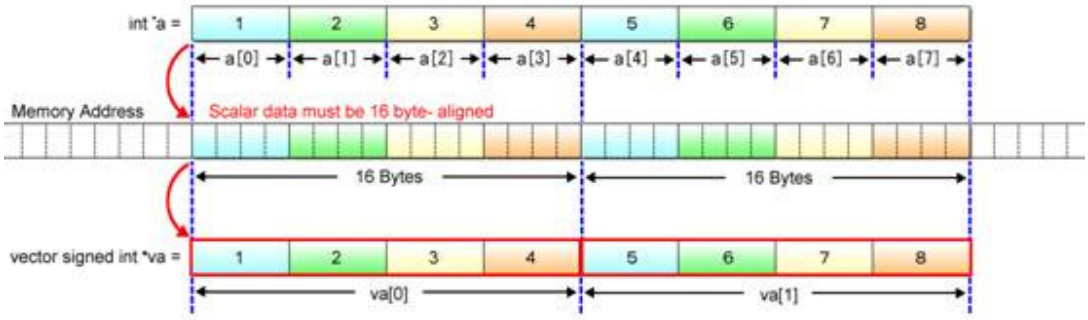


Figure 21 Byte Alignment

In Example (4), the keyword __attribute__ is used to assign the "aligned" attribute to

the scalar array *a*. The aligned attribute aligns data on the memory's byte boundary. For now, just keep in mind that data must be 16-byte aligned when referring to scalars as a vector.

Rudimentary SIMD Operations

This section describes SIMD operations using a summation as an example of simple arithmetic operations. We will compare SIMD operations with scalar operations and take a look into how data is handled, as well as how efficiently it is processed.

Program for Add Operations

Suppose that four add operations, each adding two numbers together, are required.

In such cases, conventional scalar processors use the program shown in List (1).

List (1): Scalar calculus

```
 1 int a[4] = { 1, 3, 5, 7 };
2 int b[4] = { 2, 4, 6, 8 };
3 int c[4];
4
5 c[0] = a[0] + b[0];     // 1 + 2
6 c[1] = a[1] + b[1];     // 3 + 4
7 c[2] = a[2] + b[2];     // 5 + 6
```

```
8 c[3] = c[3] + c[3];     // 7 + 8
```

Scalar operations are based on a rule of “single data = single data + single data”. That is, the four add instructions in the example must be executed in sequence as stated in the 5th to 8th lines of the program in order to obtain the sums for all four equations.

Next, let’s look at the SIMD program for the same operations.

List (2): SIMD calculus

```
1 int a[4] __attribute__((aligned(16))) = { 1, 3, 5, 7 };
2 int b[4] __attribute__((aligned(16))) = { 2, 4, 6, 8 };
3 int c[4] __attribute__((aligned(16)));
4
5 __vector signed int *va = (__vector signed int *) a;
6 __vector signed int *vb = (__vector signed int *) b;
7 __vector signed int *vc = (__vector signed int *) c;
8
9 *vc = vec_add(*va, *vb);    // 1 + 2, 3 + 4, 5 + 6, 7 + 8
```

VMX provides built-in functions for individual SIMD instructions. In the above example, **vec_add()** in the 9th line indicates the built-in function corresponding to the VMX add instruction.

SIMD operations process data on the principle of “multiple data = multiple data + multiple data”. This means that SIMD operations can process a greater quantity of data per instruction than scalar operations, making it possible to reduce the number of instructions necessary to be executed and thus the time required for processing.

Evaluator’s Comments if any:

Question 10 What are the various types of vector instruction?

Answer 10

A vector processor, or array processor, is a CPU design wherein the instruction set includes operations that can perform mathematical operations on multiple data elements simultaneously. This is in contrast to a scalar processor, which handles one element at a time using multiple instructions. The vast majority of CPUs are scalar (or close to it). Vector processors were common in the scientific computing area, where they formed the basis of most supercomputers through the 1980s and into the 1990s, but general increases in performance and processor design saw the near-disappearance of the vector processor as a general-purpose CPU.

Today, most commodity CPU designs include single instructions for some vector processing on multiple (vectorised) data sets, typically known as SIMD (Single Instruction, Multiple Data), common examples include SSE and AltiVec. Modern video game consoles and consumer computer-graphics hardware rely heavily on vector processing in their architecture. In 2000, IBM, Toshiba and Sony collaborated to create the Cell processor, consisting of one scalar processor and eight vector processors, which found use in the Sony PlayStation 3 among other applications.

Instruction types: (KSOU, 2009)

Vector – Vector instruction:

In this type, vector operands are fetched from the vector register and stored in another vector register. These instructions are denoted with the following function mappings:

F1: V -> V

F2: V*V -> V

For example, vector square root is F1 type and addition of two vectors is of two vectors is of F2.

Vector-Scalar instruction:

In this type ,when the combination of scalar and vector are fetched and stored in vector register. These instructions are denoted with the following function mappings:

F3: S*V-> V where S is the scalar item

For example, vector –scalar addition or divisions are of F3 type.

Vector reduction instructions:

When operations on vector are being reduced to scalar items as the result, then these are vector reductions instructions. These instructions are denoted with the

following function mappings:

F4:V -> S

F5: V*V -> S

For example, finding the maximum, minimum and summation of all the elements of vector are of type F4.The dot product of two vectors is generated by F5.

Vector-Memory instructions:

When vector operations with memory M are performed then these are vector-memory instructions. These instructions are denoted with the following function mappings:

F6: M -> V

F7: V-> V

Evaluator's Comments if any:

Bibliography

Al-Mouhamed, D. M. (2009). *Parallel Program Issues.* Dhahran 31261, Kingdom of Saudi Arabia: King Fahd University of Petroleum and Minerals (KFUPM) .

Angel, E. (2005). *Transformations.* Mexico: Electrical and Computer Engineering, University of Mexico.

Anonymous. (2009). *CPU cache*. Retrieved 2009, from Wikipedia: http://en.wikipedia.org/wiki/CPU_cache

Dong, Z. (2009). *Hardware of a CAD System.* British Colombia,Canada: University of Victory, Mechnical Engineering.

Foundation, F. S. (2009). *Basics of SIMD Programming*. Retrieved 2009, from Playstation 3 Linux Documents: http://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/ps3-linux-docs-08.06.09/CellProgrammingTutorial/BasicsOfSIMDProgramming.html

Hassett, D. B. (2009). *Resolving Singularities.* USA: Rice University.

HUANG, J. (Spring 02). *CS594 Visualization & Adv. Computer Graphics.* Wisconsin: University of Wisconsin.

Hwang. (2001). *Dependencies.* Omaha: University of Nebraska.

Jan, C. (2002). *Mechanisms of Motion Perception* . Irvine: University of California.

KSOU. (2009). Unit 8- Vector processor and synchronous parallel processing. In KSOU, *Adv Computer Architecture* (pp. 12-125). Delhi: Virtual Education trust.

Lee, J. (2008). *Output Primitives.* Seoul, South Korea: Seoul National University.

Lobur, L. N. (2003). Address Modes. In L. N. Lobur, *The Essentials of Computer Organization and Architecture* (p. 5.4). Sudbury, MA 01776: Jones and Bartlett Publishers.

Lovegren, J. (2007). *Instruction Sets.* San Diego, CA : San Diego State University.

Null, L. (2003). I/O Architectures. In L. Null, *The Essentials of Computer Organization and Architecture* (p. 7.3). Sudbury, MA: Jones and Bartlett Publishers.

Null, L. (2003). Cache memory. In L. Null, *The essentials of Computer Organization and Architecture* (p. 6.4). Sudbury, MA : Jones and Bartlett Publishers.

Null, L. L. (2003). RISC Machines. In L. L. Null, *The Essentials of Computer Organization and Architecture* (p. 9.2). Sudbury, MA 01776: Jones and Bartlett Publishers.

Null, L. (2003). Virtual Memory. In L. Null, *The Essentials of Computer Organization and Architecture* (p. 6.5). Sudbury, MA: Jones and Bartlett Publishers.

O'Hallaron, D. (2003). *Virtual Memory.* TROY ,NY : RENSSEALER POYTECHNIC INSTITUTE .

Shen, H.-W. (2006). *CSE581: Interactive Computer Graphics* . Ohio: The Ohio State University .

Shin, S. Y. (1998). *CS580 Computer Graphics.* South Korea: Korea Advanced Institute of Science and Technology.