Syllabus for Computer Architecture and Parallel Programming, CS 2420

Instructor: Dr. Mark Fienup, Computer Science Department
Office Location: 313ITT
Office Phone: 319-273-5918
Email Address: fienup@cs.uiue.edu
Office Hours: M: 8-11:45, 1:10-3, T: 9:30-10:45, 1:10-2, W: 1:10-2, Th: 9:30-10:45, 1:10-2, F: 8-10

Meeting Time & Location:
Course meets Tuesday and Thursday from 2 PM to 3:15 PM in ITT 328

Course Information

Course Description:
Computer architecture of uniprocessor and multiprocessor systems with an emphasis on parallel programming to achieve high performance. Topics include processor design (pipelining and superscalar), memory hierarchy, interconnection networks, performance metrics, parallel program design, and parallel programming tools for multi-core systems, clusters, and graphics processing units (GPUs)

Prerequisites: Introduction to Computing (CS 1510) and Computer Organization (CS 1410)
The ability to program in a high-level programming language (e.g., Python, Java, C/C++, Ada, etc.) is necessary to succeed in this course. Successful completion of your Introduction to Computing course should provide you with sufficient programming ability. Knowledge of uniprocessor computer organization including components (CPU, memory, I/O devices, system bus, etc.), fetch-decode-execution cycle, and some understanding of assembly language programming are needed from your Computer Organization course.

Purpose of the course:
The purpose of this course is to enable you to write software that can utilize the computational power of current and future computer architectures. Your current skills only allow you to write single-threaded programs which utilize only a single processor core. This might be okay if the computer only has 2 or 4 cores, but as the number of cores per processor increases your future employer will want you to do better. Already computers have programmable Graphics Processing Units (GPU, i.e., graphics cards) with thousands of processors, and computers are networked together into clusters, we'll also learn how to write programs that leverage these technologies to vastly increase the computational power of your programs.

Course Organization:
This course is taught as a “flipped classroom.” That means you'll have textbook reading, video to watch, quiz to take, and questions to answer before each class. The video will cover the "easier" material for the class so we can spend in-class time focusing on the more challenging material.

The course is organized into five units:

1. Uniprocessor computer architecture and C programming – Before you can write efficient parallel programs, you must first learn how to design and write efficient uniprocessor programs. Understanding the computer architecture (i.e., pipelining and superscalar processor design and memory hierarchy) are key to achieving high performance. As you learn about computer architecture, you will also be learning to program in C which will be used to write sequential and parallel programs throughout the course. (Weeks 1 to 3)
2. Parallel hardware and parallel software design – Before you can design and write efficient parallel programs, you must also understand the parallel hardware on which your parallel programs will execute. As you learn about parallel hardware, you will also be learning general parallel program design techniques and patterns. (Weeks 4 and 5)
3. Shared-memory programming with pthreads – For a relatively small number of processors (e.g., current multi-core computers), shared-memory design is common. A parallel program runs on multiple processors/cores as "light-weight" (i.e., low overhead points of execution in the same program) threads. These threads communicate data and synchronize tasks by reading and writing data in the shared memory. You will practice your parallel program design skills by writing several C program that utilize the pthread module to create and coordinate threads running on multiple processor cores. (Weeks 6 to 9)
4. Distributed-memory programming with MPI – As the number of processors/cores increase, they all contend for memory-bandwidth. Since memory holds their running programs and data, memory becomes a bottleneck (von Neumann bottleneck). Thus, all large parallel computers are distributed-memory machines, i.e., a collection of networked, “stand-alone” computers each with their own local memory. A parallel program running across multiple computers needs to communicate data and synchronize tasks using the network. You will practice your parallel program design skills by writing several C program that utilizing MPI (Message-Passing Interface) functions which is the standard distributed-memory programming tool. (Weeks 10 to 12)
5. General-purpose GPU programming with CUDA – Driven by 3D computer games, graphics accelerator cards (i.e., GPU—Graphics Processing Unit) where developed with thousands of processors to perform mathematically intense real-time 3D graphical rendering. To leverage the computational power of GPUs for general purpose (i.e., non-graphics) computing tasks, NVIDIA modified their GPU architecture to create the CUDA Architecture about 2007. You will practice your parallel program design skills by writing several C program that utilizing CUDA extensions to execution on massively parallel GPU. (Weeks 13 to 15)

Textbooks and Course Packet

Required: An Introduction to Parallel Programming, Peter S. Pacheco, 2011, 1st Edition, Morgan Kaufmann Publishers,

Computer Architecture (CS 2420 - Fienup) Course Packet – available at CopyWorks (2227 College Street, Cedar Falls, IA 319 266-2306)

Course Learning Objectives

Learning Objective #1: Explain the operation of uniprocessor computer components including the processor (pipelined and superscalar) and memory hierarchy (cache and virtual memory).

Learning Objective #2: Demonstrate an understanding of uniprocessor computer architecture by designing and writing C programs that make efficient use of the processor and memory hierarchy.

Learning Objective #3: Explain the operation of parallel hardware including cache-coherence and mutexes on shared-memory machines, and interconnect performance (bisection bandwidth, bandwidth and latency) characteristics on distributed-memory machines.

Learning Objective #4: Demonstrate an understanding of parallel hardware and general parallel program design techniques and patterns by producing efficient parallel program designs to minimize parallel program overhead.

Learning Objective #5: Demonstrate an understanding of shared-memory machines by designing and writing C programs using the pthread module that make efficient use of multiple cores.

Learning Objective #6: Demonstrate an understanding of distributed-memory machines by designing and writing C programs using the MPI module that make efficient use of multiple networked processors.

Learning Objective #7: Demonstrate an understanding of GPU programming by designing and writing CUDA programs that make efficient use of the GPU processing power.

Course Requirements

Instructional Methods / Activities / Assessments

This course consists of the following activities and assessments to assist you in achieving the course and unit objectives. For consistency and planning purposes, each week is structured the same and consists of the following:

<table>
<thead>
<tr>
<th>Activity/Assessment</th>
<th>Due Date and Time</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Textbook Reading</td>
<td>Before the Tuesday Video</td>
<td>You can do these in any order that you find helpful. Often the video will provide more depth or further explanation of the textbook’s coverage.</td>
</tr>
<tr>
<td>&quot;Tuesday&quot; Video</td>
<td>Before the &quot;Tuesday&quot; Online Quiz</td>
<td></td>
</tr>
<tr>
<td>&quot;Tuesday&quot; Video Online Quiz</td>
<td>Tuesday at 2 PM</td>
<td>Automatically graded eLearning quiz—typically 5 multiple choice questions</td>
</tr>
<tr>
<td>Lab Video</td>
<td>Mid-week/After Tuesday video</td>
<td>Zip together programs and answers to written questions. Graded by instructor using the Lab Rubric.</td>
</tr>
<tr>
<td>Lab .zip submission</td>
<td>Saturday at 11 PM</td>
<td></td>
</tr>
<tr>
<td>Textbook Reading 2</td>
<td>Before the Thursday Video</td>
<td>You can do these in any order that you find helpful. Often the video will provide more depth or further explanation of the textbook’s coverage.</td>
</tr>
<tr>
<td>&quot;Thursday&quot; Video</td>
<td>Before the Thursday Online Quiz</td>
<td></td>
</tr>
<tr>
<td>&quot;Thursday&quot; Video Online Quiz</td>
<td>Thursday at 2 PM</td>
<td>Automatically graded eLearning quiz—typically 5 multiple choice questions</td>
</tr>
<tr>
<td>Weekly Homework Assignment</td>
<td>The following Wednesday at 5 PM (HWs 11 &amp; 12 are due in two weeks)</td>
<td>The first 5 HWs will consist of written and problem based questions. The remaining HWs will be programs and graded using the Homework Rubric.</td>
</tr>
</tbody>
</table>

Grading

There are a total of 650 points possible across all assessment activities. The following table shows a breakdown of these points.

<table>
<thead>
<tr>
<th>Assessment Activity</th>
<th>Number of Assessments</th>
<th>Points per Assessment</th>
<th>Total for Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weekly Labs</td>
<td>14 (no week 15 lab)</td>
<td>10 points each</td>
<td>140 points</td>
</tr>
<tr>
<td>Homework Assignments</td>
<td>12 (HWs 11 &amp; 12 are two weeks)</td>
<td>20 points each</td>
<td>240 points</td>
</tr>
<tr>
<td>Online Video Quizzes</td>
<td>30</td>
<td>5 points each</td>
<td>150 points</td>
</tr>
<tr>
<td>Unit Quizzes: Units 1 to 4</td>
<td>4</td>
<td>15 points each</td>
<td>60 points</td>
</tr>
<tr>
<td>Final Exam — includes Unit 5</td>
<td>1</td>
<td>60 points</td>
<td>60 points</td>
</tr>
<tr>
<td><strong>Total Points</strong></td>
<td><strong>650 points</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Your course grade will be based by the percentage of these 650 points that you accumulate. Grades will be assigned based on straight percentages off the top student score. If the top student’s percentage is 94%, then the grading scale will be 94-100% A’s, 83.9-94% B’s, 73.9-83% C’s, 63.9-73% D’s and below F’s. Plus and minus grades will be assigned for students near cutoff points.

Technology Requirements:

The following information has been provided to assist you in preparing to use technology successfully in this course. You may need to install free software on your computer. You will need the following:

- Internet access/connection – high speed recommended (not dial-up)
- Email – mostly to ask questions asynchronously
- Word Processor (e.g., Open-office, MS Word, etc.)
- SSH and telnet client (free options for Windows: PuTTY from [http://www.putty.org/](http://www.putty.org/); MAC or Linux users can use: ssh userName@computerName in a terminal window)
- Secure file transfer client (free options for Windows: FileZilla from [https://filezilla-project.org/](https://filezilla-project.org/); MAC or Linux users can use: scp -r userName@computerName:remoteDir localDir)
Communication and Support:
Even though this is a Face-to-Face course, you will need to communicate with the instructor and fellow students using several forms:
- Face-to-Face meeting during office hours (listed above) are my preferred form of communication on technical course material. You can come to my office or arrange a Google Hangout.
- Email—email should be used if you have personal concerns or questions. I will typically respond within a couple of hours, but maybe as longer at night or on weekends. If several students ask the same question, I will post an answer to the Frequently Asked Questions (FAQ) eLearning Discussion Board.
- eLearning Announcements—periodically I will post eLearning Announcements to alert you to important course related information.

Grading response time
Students will have immediate feedback on online quiz grades. For labs and homework assignments, I will make every attempt to grade and post those grades within a week after their submission time has expired. Sometimes, however, other university work commitments may delay the grading process.

Technical Support
If you experience any technical problem do not contact the instructor. Please contact the Office of Continuing and Distance Education at 319-273-7740 or cesp-consult@uni.edu. Support hours are available Tuesday—Thursday 8am—9pm and Thursday 8am—5pm.

Course and University Policies:
Late Assignment Policy
Any type of assignment not submitted by its submission time is considered late. Late assignments will be accepted for up to one week after the submission time, but will incur a 25% penalty.

Honor Code
Copying from other students is expressly forbidden. Doing so on assignments will be penalized every time it is discovered. The penalty can vary from zero credit for the copied items (first offense) up to a failing grade for the course. If an assignment makes you realize you don’t understand the material, ask questions designed to improve your understanding, not ones designed to discover how another student solved the assignment. The solutions to assignments should be individual, original work unless otherwise specified. Remember: discussing assignments is good. Copying code or question answers is cheating.

Any substantive contribution to your assignment solution by another person or taken from a publication (or the web) should be properly acknowledged in writing. Failure to do so is plagiarism and will necessitate disciplinary action. Additionally, assisting or collaborating on cheating is cheating. Cheating can result in failing the course and/or more severe disciplinary actions. You are responsible for being familiar with complete set of University Academic Ethics Policies (http://www.uni.edu/pres/policies/301.shtml).

Academic Learning Center Services
You are encouraged to utilize the Academic Learning Center’s free assistance with writing, reading, and learning strategies. UNI’s Academic Learning Center is located in C008 ITTC. Visit the website at http://www.uni.edu/unialc or phone 319-273-2361 for more information.

Office of Student Disability Services
The Americans with Disabilities Act of 1990 (ADA) provides protection from illegal discrimination for qualified individuals with disabilities. Students requesting instructional accommodations due to disabilities must arrange for such accommodation through the Office of Student Disability Services (SDS). The SDS is located at: 103 Student Health Center, the phone number is: (319) 273-2677, and the website is http://www.uni.edu/sds/.

UNI Academic Ethics/Discipline Policy
Students at the University of Northern Iowa are required to observe the commonly accepted standards of academic honesty and integrity. http://www.uni.edu/policies/301

UNI Student Code of Conduct Policy
The university’s student conduct code maintains the principles of respect, honesty, and responsibility to create a safe, healthy environment for members of the campus community while preserving an educational process that is consistent with the mission of the University. http://www.uni.edu/policies/302
Week 1 Discussion Questions
Chapter 1:

1. Why have processor designers moved from single-core to multicore designs?
2. Why can’t ordinary serial programs utilize multiple cores?
3. What is the difference between task-parallelism and data-parallelism?
4. Define each of the following parallel programming terms related to coordinating work of multiple cores:
   a) communication
   b) load balancing
   c) synchronization
5. What are the main characteristics of each type of parallel system?
   a) shared-memory system
   b) distributed-memory system

Chapter 2.1:

6. What is the function of each of the following CPU components?
   a) program counter (PC)
   b) instruction register (IR)
   c) register file
   d) arithmetic and logic unit (ALU)

7. What is meant by the term von Neumann bottleneck?
8. How do even single-core operating systems do multitasking?
9. When a program runs (i.e., a process) what is stored in each part of its main memory?
   a) call stack/run-time stack
   b) global data area
   c) heap
   d) program/text area
10. What is the difference between a process and a thread?
**Objective:** Demonstrate your ability to edit, compile, run a simple C program while capturing its output to a file.

### Directory Navigation and Listing

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cd</td>
<td>change to home directory</td>
</tr>
<tr>
<td>cd ..</td>
<td>go up to parent directory</td>
</tr>
<tr>
<td>cd subdir</td>
<td>change to subdirectory subdir</td>
</tr>
<tr>
<td>ls</td>
<td>list content of current directory</td>
</tr>
<tr>
<td>ls -l</td>
<td>list content with details</td>
</tr>
<tr>
<td>ls -a</td>
<td>list content including hidden files</td>
</tr>
</tbody>
</table>

### Process Management

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ps -aux</td>
<td>grep uname List processes for uname</td>
</tr>
<tr>
<td>top</td>
<td>Shows the real-time processes</td>
</tr>
<tr>
<td>kill -9 pid</td>
<td>Kills the process with pid #</td>
</tr>
</tbody>
</table>

### Keyboard Shortcuts

<table>
<thead>
<tr>
<th>Key Combination</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;tab&gt;</td>
<td>Auto-complete partial file name</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;+c</td>
<td>Kill current command/program</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;+z</td>
<td>Sleep current program</td>
</tr>
<tr>
<td>&lt;↑&gt;</td>
<td>Recall previous command(s)</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;+d</td>
<td>log-off and close terminal</td>
</tr>
</tbody>
</table>

### File Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp src dest</td>
<td>copy src file to dest file</td>
</tr>
<tr>
<td>cp -r sDir dDir</td>
<td>copy &quot;recursively&quot; sDir directory to dDir directory (copies subdirectories too)</td>
</tr>
<tr>
<td>mv src dest</td>
<td>move - renames src as dest</td>
</tr>
<tr>
<td>rm fileName</td>
<td>removes file fileName</td>
</tr>
<tr>
<td>rm -r dirName</td>
<td>removes directory recursively</td>
</tr>
<tr>
<td>rmdir dirName</td>
<td>removes empty dirName</td>
</tr>
<tr>
<td>mkdir dirName</td>
<td>makes directory called dirName</td>
</tr>
<tr>
<td>chmod 750 file1</td>
<td>change permission of file1 by specifying a three digit octal # where digits are owner, group, world each octal digit in binary are: read (4), write (2), execute (1)</td>
</tr>
<tr>
<td>cat file1</td>
<td>display file1 to screen</td>
</tr>
<tr>
<td>less file1</td>
<td>display file1 with pagination (space - next page, q-exit, ↑,↓-keys)</td>
</tr>
</tbody>
</table>

### "Programming" Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nano file.c</td>
<td>Simple text-editor</td>
</tr>
<tr>
<td>emacs file.c</td>
<td>Better C/C++ editor (see emacs handout)</td>
</tr>
<tr>
<td>gcc file.c</td>
<td>C compiler: compile to a.out</td>
</tr>
<tr>
<td>g++ file.cpp</td>
<td>C++ compiler: compile to a.out Options: compile to exeFile instead</td>
</tr>
<tr>
<td>-o exeFile</td>
<td></td>
</tr>
<tr>
<td>./a.out</td>
<td>execute program in current directory (&quot;.&quot;) called a.out</td>
</tr>
<tr>
<td>time exeFile</td>
<td>run exeFile and print timing when done</td>
</tr>
<tr>
<td>script out.txt</td>
<td>capture output to file out.txt</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;+d to end</td>
<td></td>
</tr>
</tbody>
</table>

1) Log-on to student.cs.uni.edu using a Telnet/ssh client (e.g., PuTTY: http://www.chiark.greenend.org.uk/~sgtatham/putty/)
   (On a MAC or in Linux you can probably use: ssh userName@student.cs.uni.edu in a terminal window to log-on)

2) Your initial log-in is the same as your UNI CatID with initial password of: 1234temp

3) For this activity I want you to:
   - create and then move into a directory called lab1 to store files for this assignment
   - use an editor (emacs or nano) to write a simple C program (on next back page) that prompts the user for their name and age, allows them to enter it, and outputs it back for them. Use the file name age.c
   - compile the C program to an executable file called age using: gcc -o age.age.c
   - when its working, capture the interactive running of the program using: script out.txt to start the capture, ./age to run the program, and <Ctrl>+d to end the capture
   - display the contents of the out.txt to the screen using the less out.txt command (q-to exit less)
4) Use a secure ftp client (e.g., FileZilla: https://filezilla-project.org) to copy the directory lab1 to your local computer
   (On a MAC you can probably use: scp -r userName@student.cs.uni.edu:hw4 localDir)

5) On your local computer, zip the lab1 directory and submit lab1.zip on the eLearning system as Lab #1.

```c
/* File: age.c
   Compile by: gcc -o age age.c
   Run by: ./age
*/

#include <stdlib.h>
#include <stdio.h>

const int SIZE = 100;

int main(int argc, char * argv[]) {
   char name[SIZE];
   int age;

   printf("Enter your name: ");
   scanf("%s", name);

   printf("Enter your age: ");
   scanf("%d", &age);

   printf("%s your age is %d.\n", name, age);

   return 0;
} // end main
```
Emacs Reference

Starting Emacs
emacs filename

Command Keys Meaning
C-key Hold <Ctrl>-key down while hitting specified “key”
A-key Hold <Alt>-key down while hitting specified “key”

Exit Emacs and File Commands
C-x C-c Exit emacs-- answer ‘y’ to save work back to file
C-x C-s Save work back to disk without exiting
C-x C-w Write work to a file that you specify

Cursor Motion - arrow keys: ↑ ↓ ← →
C-a Move to beginning of the current line
C-e Move to end of the current line
C-v Move down a “page”
A-v Move up a “page”
A--< Move to beginning of the file
A-> Move to end of the file
A-g g # Go to specified line #

Editing - Cutting, pasting, search, replace
C-k Kill/cuts from cursor up to end of line. Puts line in kill buffer
C-y Yank/paste copy of kill buffer at cursor location
C-spacebar Begin marking at cursor location
C-w Wipe/delete from beginning mark to cursor location. Puts deleted text into kill buffer
A-w Copy without deleting from beginning mark to cursor location. Puts copied text into kill buffer
C-c C-c Comment out from beginning mark to cursor location
C-u C-c C-c Uncomment out marked region
C-s Interactive Search as you type search string at bottom of screen. Hit C-s again to repeat last search (wraps to top)
C-r Interactive Reverse Search (“upward” from cursor)
A-a Query replace. At bottom of screen type search-string<enter> followed by replacement-string<enter>. Highlights first occurs after cursor. Hit:
  <spacebar> to replace and move to next match
  ! to replace all remaining matches without questions
  <backspace> to skip to next occurrence
  <Enter> to quit

Windows, Buffers, and Regions
C-x 2 Split window into two buffers vertically
C-x 3 Split window into two buffers horizontally
C-x o Move cursor to other buffer
C-x 1 Collapse window into one containing cursor
C-x c-f Open or create file in window containing cursor
Learning Objectives:

- Trace the execution of a high-level language program (process) with respect to the fetch-decode-execute cycle, run-time stack, and parameter passing.
- Explain how single-core computers can multitask multiple processes to achieve simultaneous execution.
- Identify task-parallelism and data-parallelism in a non-programming situation.

1. Trace the execution of calling Power with num = 3 and pow = 4 by showing the run-time stack on the diagram on the next page.

2. Consider the following scenario on a single-core system:

- four processes A, B, C, and D are all loaded into memory and ready to execute (i.e., in the ready queue),
- process A is selected by the OS to run, but after 1 μs requests to read from a file on disk 1
- process B is selected by the OS to run, but after 2 μs requests to read from a different file on disk 1

a) Referring to the Hardware Support for the OS Supplement in which queue would each process reside?

b) On page 18 of the textbook is correctly states:

"A multitasking OS may change the running process many times a minute, even though changing the running process can take a long time."

What steps would the OS need to perform to change between running process B and process C?

c) Assuming that process A’s read from a file takes a long time (say 20 ms), how is the reading of the file performed at the same time as the execution of process B?

d) How would process A get put back into the Ready queue of the OS?

3. Exercise 1.6 on pages 13-14 of the textbook.

4. Exercise 1.8 on page 14 of the textbook.

1.6 Derive formulas for the number of receives and additions that core 0 carries out using

   a. the original pseudo-code for a global sum, and
   b. the tree-structured global sum.

   Make a table showing the numbers of receives and additions carried out by core 0 when the two sums are used with 2, 4, 8, ..., 1024 cores.

1.8 Suppose the faculty are going to have a party for the students in the department.

   a. Identify tasks that can be assigned to the faculty members that will allow them to use task-parallelism when they prepare for the party. Work out a schedule that shows when the various tasks can be performed.

   b. We might hope that one of the tasks in the preceding part is cleaning the house where the party will be held. How can we use data-parallelism to partition the work of cleaning the house among the faculty?

   c. Use a combination of task- and data-parallelism to prepare for the party. (If there’s too much work for the faculty, you can use TAs to pick up the slack.)
Homework #1

main:

maxNum = 3
maxPower = 4

CalculatePowers(maxNum, maxPower)

(*)

...
end main

CalculatePowers(In: integer numLimit,
               integer powerLimit)

integer num, pow, result

for num := 1 to numLimit do
    for pow := 1 to powerLimit do
        Power(num, pow, result)

(**) print num " raised to " pow " power is "
    result
    end for pow
end for num
end CalculatePowers

Power(In: integer n, integer e, Out: result)
if e = 0 then
    result = 1
else if e = 1 then
    result = n
else
    Power(n, e - 1, result)
    result = result * n
end if
end Power

Run-time Stack

Due: Wednesday, August 31 at 5 PM
An efficient parallel implementation of a serial program may not be obtained by finding efficient parallelizations of each of its steps. Rather, the best parallelization may be obtained by stepping back and devising an entirely new algorithm.

As an example, suppose that we need to compute \( n \) values and add them together. We know that this can be done with the following serial code:

```plaintext
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute.next_value( . . );
    sum += x;
}
```

Now suppose we also have \( p \) cores and \( p \) is much smaller than \( n \). Then each core can form a partial sum of approximately \( n/p \) values:

```plaintext
my_sum = 0;
my.first.i = . . . ;
my.last.i = . . . ;
for (my.i = my.first.i; my.i < my.last.i; my.i++) {
    my.x = Compute.next_value( . . );
    my_sum += my.x;
}
```

Here the prefix \( my. \) indicates that each core is using its own, private variables, and each core can execute this block of code independently of the other cores.

After each core completes execution of this code, its variable \( my\_sum \) will store the sum of the values computed by its calls to \( Compute\_next\_value \). For example, if there are eight cores, \( n = 24 \), and the 24 calls to \( Compute\_next\_value \) return the values

```
0 1 2 3 4 5 6 7
1 4 3 9 2 8 5 1 1 6 2 7 2 5 0 4 1 8 6 5 1 2 3 9
```

then the values stored in \( my\_sum \) might be

```
0 1 2 3 4 5 6 7
1 9 8 19 7 15 17 13 12 14
```

Here we're assuming the cores are identified by nonnegative integers in the range \( 0,1, . . ., p - 1 \), where \( p \) is the number of cores.

When the cores are done computing their values of \( my\_sum \), they can form a global sum by sending their results to a designated "master" core, which can add their results:

```plaintext
if (I'm the master core) {
    sum = my.x;
    for each core other than myself {
        receive value from core;
        sum += value;
    }
} else {
    send my.x to the master;
}
```
FIGURE 1.1
Multiple cores forming a global sum
Build house

1. hole
2. pour foundation
3. floor
4. frame walls
5. roof
6. sliding

CPU/Processor

Control Unit

PSW - process status word

ALU (Add, Subtract, etc.)

R0  R1  R2  R3

Register File

R31

PC, program counter

IR, instruction register

MAR, memory addr. reg

MBR, memory buffer reg

System Bus

Memory

Addresses

0  1  2  3

Unused

Stack

Heap

Global Address Space

A Program's Address Space

Load R3, Y
Load R2, X
Add R1, R2, R3
Store R1, SUM

Operating System Area

Program Area

I/O Controller

Disk

Processing (Instruction/Machine) Cycle of stored-program computer - repeat all day
1. Fetch Instruction - read instruction pointed at by the program counter (PC) from memory into Instruction Reg. (IR)
2. Decode Instruction - figure out what kind of instruction was read
3. Fetch Operands - get operand values from the memory or registers
4. Execute Instruction - do some operation with the operands to get some result
5. Write Result - put the result into a register or in a memory location
(Note: Sometime during the above steps, the PC is updated to point to the next instruction.)
<table>
<thead>
<tr>
<th>Type of Instruction</th>
<th>MIPS Assembly Language</th>
<th>Register Transfer Language Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Access (Load and Store)</td>
<td>lw $4, Mem</td>
<td>$4 ← [Mem]</td>
</tr>
<tr>
<td></td>
<td>sw $4, Mem</td>
<td>Mem ← $4</td>
</tr>
<tr>
<td></td>
<td>lw $4, 16($3)</td>
<td>$4 ← [Mem at address in $3 + 16]</td>
</tr>
<tr>
<td></td>
<td>sw $4, Mem</td>
<td>[Mem at address in $3 + 16] ← $4</td>
</tr>
<tr>
<td>Move</td>
<td>move $4, $2</td>
<td>$4 ← $2</td>
</tr>
<tr>
<td></td>
<td>li $4, 100</td>
<td>$4 ← 100</td>
</tr>
<tr>
<td>Load Address</td>
<td>la $5, mem</td>
<td>$4 ← load address of mem</td>
</tr>
<tr>
<td>Arithmetic Instruction (reg. operands only)</td>
<td>add $4, $2, $3</td>
<td>$4 ← $2 + $3</td>
</tr>
<tr>
<td></td>
<td>mul $10, $12, $8</td>
<td>$10 ← $12 * $8 (32-bit product)</td>
</tr>
<tr>
<td></td>
<td>sub $4, $2, $3</td>
<td>$4 ← $2 - $3</td>
</tr>
<tr>
<td>Arithmetic with Immediates (last operand must be an integer)</td>
<td>addi $4, $2, 100</td>
<td>$4 ← $2 + 100</td>
</tr>
<tr>
<td></td>
<td>mul $4, $2, 100</td>
<td>$4 ← $2 * 100 (32-bit product)</td>
</tr>
<tr>
<td>Conditional Branch</td>
<td>bgt $4, $2, LABEL</td>
<td>Branch to LABEL if $4 &gt; $2</td>
</tr>
<tr>
<td></td>
<td>(bge, bhi, beq, bne)</td>
<td></td>
</tr>
<tr>
<td>Unconditional Branch</td>
<td>j LABEL</td>
<td>Always Branch to LABEL</td>
</tr>
</tbody>
</table>

Fibonacci Sequence: 0 1 1 2 3 5 8 13 21
Position in Sequence: 0 1 2 3 4 5 6 7 8

A high-level language program to calculate the $n^{th}$ fibonacci number would be:

```plaintext
temp2 = 0
temp3 = 1
for i = 2 to n do
    temp4 = temp2 + temp3
    temp2 = temp3
temp3 = temp4
end for
result = temp4
```

A complete assembly language MIPS program to calculate the $n^{th}$ fibonacci number.

```assembly
.data
n:       .word 8  # variable in memory
result:  .word 0  # variable in memory

.text
.globl main
main:    li $2, 0  # $2 holds temp2
         li $3, 1  # $3 holds temp3
for_init: li $6, 2  # initialize i ($6) to 2
          lw $5, n  # load "n" into $5
for_loop: bgt $6, $5, end_for  # if $6 >= $5, then branch to end_for label
          add $4, $2, $3  # $4 holds temp4
          move $2, $3  # shift temp3 to temp2
          move $3, $4  # shift temp4 to temp3
          addi $6, $6, 1  # increment i ($6)
          j for_loop  # unconditionally jump to for_loop label
end_for:  sw $4, result  # store the result to memory
          li $v0, 10  # system call for exit
syscall  # call the operating system
```

W1F Page 2
Hardware Support for Operating System Supplement

What is an operating system (OS)?

- A program that operates as the interface between the user and the hardware

<table>
<thead>
<tr>
<th>Runs in User Mode</th>
<th>Web Browser</th>
<th>Accessing package</th>
<th>etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compiler</td>
<td>Editors</td>
<td>Command Interpreter</td>
</tr>
<tr>
<td></td>
<td>Window system</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Runs in Kernel or Supervisor Mode</th>
<th>Operating System - file system, memory manager, etc.</th>
</tr>
</thead>
</table>

Goals of OS

1. Make computer convenient to use by providing a virtual/extended machine that is easier to use and program than the underlying hardware,
   e.g., writing/reading to file on disk

   ![I/O Controller Diagram]

   OS provides high-level system calls so programmer does not need to know details of disk

2. Use computer resources/hardware efficiently
   Resources - processor(s), memory, timers, disks, network

Resources are competed for by all running programs

Examples:
- which programs are loaded in limited memory
- restricts access to memory used by other programs and the operating system
- which program can run on the CPU

We can view the OS as resource manager that is responsible for resource allocation, tracking resources, accounting, and mediating conflicting requests
Hardware Support for Operating System Supplement

**OS manages processes (running programs):**
A *process* is the term for a running program. A process's state consists of the CPU register values, its runtime stack in memory, and its other memory content. Many processes may be executing concurrently, but only one can be executing on a CPU at a time. When the CPU switches to another process, a context switch occurs which involves saving the complete state of the previously executing process before loading the state of the next process to execute into the CPU. Depending on the hardware, this can take up to 1000 microseconds (i.e., very slow in computer terms).

![Process State Diagram]

Queues are used to hold *process control blocks (PCB)* that represent processes internally to the OS.

![Process Control Block]

---

Hardware Support for OS Supplement - Page 2
OS maintains queues and does scheduling:

The PCB for a process moves around from queue to queue depending on its state.

I/O queues - since I/O is so slow, several programs might have outstanding requests to use an I/O device so a queue for each I/O device is necessary.

Ready (Short-term) queue - programs that are in memory and ready to execute. All they need is the CPU to run.

Medium-term queue - programs that are partially executed, but have been swapped out of memory to disk

Long-term queue - user has requested that a program be executed, but it has not yet been loaded into memory

Hardware support for Operating Systems

Need protection from user programs that:
1. go into an infinite loop
2. access memory of other programs or the OS
3. access files of other programs

Protection Techniques

1) Dual-Mode Operation - the CPU has two (or more) modes of operation: user mode and system (supervisor/monitor/privileged) mode with some privileged (machine/assembly language) instructions only executable in system mode. A mode-bit within the CPU's processor-status-word (PSW) register is used to indicate whether the CPU is executing in user or system mode. The set of all machine-language instructions are divided into:
a) privileged instructions that can only be executed in system mode, and
b) non-privileged instructions that can be executed in any mode of operation.

Every time an instruction is executed by the CPU, the control-unit hardware checks to see if the instruction is privileged and whether the mode is user. Whenever this case is detected, an exception (internal interrupt) is generated that turns CPU control back over to the operating system.
Hardware Support for Operating System Supplement

CPU Timer - the operating system sets a count-down timer before turning control over to a user program. If the timer expires, it generates an interrupt before the user program is started. Remember that only one program (in a single CPU system) can be executing at a time so when the OS turns control over to a user program it has “lost control.”
Modifications to the CPU timer are privileged.

2) Restrict a user program to its allocated address space in memory. In a simple computer, a user program might be allocated a single contiguous address space in memory. The two special purpose CPU registers: StartMemory and EndMemory can bracket the user program’s address space. All memory addresses that the user program performs can be checked by hardware in the CPU to make sure that they fall between the values in these registers. If the user program tries to access memory outside the range of addresses indicated by these registers, an interrupt/exception is raised to return control back to the operating system. On more complex computers, a memory-management unit (MMU) provides a more sophisticated address mapping scheme (paging, segmentation, paged segments, none). Modifications to the memory-management registers are privileged.

3) Protection to restrict a process from accessing files of other programs varies depending on whether the computer is using memory-mapped I/O or instruction-based I/O (see section 4.5 of text). If memory-mapped I/O is being used, the memory address associated with the external device I/O registers are outside of the process accessible memory address space. Thus, our solution (2) above is enough to force a process to request I/O through operating system calls.

If instruction-based I/O instructions are being used. I/O has a separate address space from memory, but we can make these I/O instructions privileged so they can only be executed in system mode. Thus, a user process could not execute them directly.
Week 2 Discussion Questions
Chapter 2.2.1 - 2.2.3:

1. Why are L1 caches not implemented using fully-association?
2. What is meant by the term cache conflict?
3. What is the advantage of a 4-way set-associative cache over a direct-mapped cache of the same size?
4. What is the advantage of a direct-mapped cache over a 4-way set-associative cache of the same size?
5. Exercise 2.3 on page 77.

Chapter 2.3.4:

6. Exercise 2.4 on page 77.
7. What are the goal(s) of virtual-memory?
8. On a virtual-memory system, where is the page-table stored?
9. What is contained in the TLB (translation-lookaside buffer)?
10. There are many similarities between the cache-to-main memory level and the main memory-to-disk level of the memory hierarchy, since the main-memory is used as a cache of papers for the slower disk. However, one important difference is the action taken when a process has a cache miss vs. when a process has a page fault:
    - On a cache miss, the process stalled the CPU while the slower main memory is accessed.
    - On a page fault, the CPU is taken away from the process while the slower disk is accessed.

Why are these situations treated differently?
Learning Objectives:
- Write correct programs using pointers to pass parameters to and from subprograms.
- Write correct programs that dynamically allocate arrays.

To start the lab:
- watch the Lab 2 Video on the eLearning system
- download lab2.zip from the eLearning system and unzip/extract it locally on your computer
- copy the lab2 directory to student.cs.uni.edu using a secure ftp client (winSCP, FileZilla, scp, etc.)
- log-on to student.cs.uni.edu using Putty/ssh

Part A: Using an editor on student.cs.uni.edu open the file displayCircleInfo.c which contains a simple C program to interactively allow the user to enter a circle’s radius and it outputs the circles area and circumference. Notice that the program is split into a main function that acts as the high-level outline for the program and calls three functions: getRadius, calculateAreaAndCircumference, and displayCircleInfomation.

Answer the following questions about the displayCircleInfo.c program:
The main calls calculateAreaAndCircumference:
calculateAreaAndCircumference(radius, &area, &circumference);
a) What is the purpose of & symbol in front of the area, and circumference parameters?

b) Why does the radius parameter not have a & symbol in front of it?

The calculateAreaAndCircumference function definition is:
void calculateAreaAndCircumference(double radius, double * area,
double * circumference) {
    *area = PI * pow(radius, 2.0);
    *circumference = 2.0 * PI * radius;
} // end calculateAreaAndCircumference
c) What is the purpose of * symbol in front of the area, and circumference parameters in the function header?

d) What is the purpose of * symbol in front of the area and circumference in the assignment statements?

The getRadius function definition is:
double getRadius() {
    double radius;
    printf("Enter the radius of a circle: ");
    scanf("%lf", &radius); // NOTE %lf for double, but %f for float
    return radius;
} // end getRadius
e) What is the purpose of double in the double getRadius() { function header?

f) What is the purpose of & symbol in front of the radius parameter in the scanf call?
Part B: Write a similar program called displayRectangleInfo.c that is split into a main function that acts as the high-level outline for the program and calls three functions:
- getDimensions - prompts the user and return the length and width of the rectangle,
- calculateAreaAndCircumference - calculates and returns rectangle’s area and circumference, and
- displayRectangleInformation - displays the rectangle's information to the user in a nicely formatted fashion.
Write, compile, debug your program on student.cs.uni.edu. Use the script command to capture the user interaction when you run the final version.

Part C: Using an editor on student.cs.uni.edu open the file averageScores.c which contains a simple C program to interactively allow the user to enter a collection of scores to be averaged.
Answer the following questions about the averageScores.c program:

a) What is the maximum number of scores that can be handled by this program?

The main calls getScores as:
```c
getScores(&numberOfScores, scores);
```

b) What is the purpose of & symbol in front of the numberOfScores parameter?

c) Why does the scores parameter not need a & symbol in front it?

The getScores function definition is:
```c
void getScores(int * count, double scores[])
{
    double score;
    printf("Enter scores one at a time (enter -1 to quit.)\n");
    *count = 0;
    while (1) { // infinite loop any nonzero integer is True
        printf("Enter a score (or -1 when done): ");
        scanf("%lf", &score);
        if (score < 0.0) {
            break;
        } // end if
        scores[*count] = score;
        (*count)++;
    } // end while
} // end getScores
```
d) Why are the paranthesis necessary when incrementing the count? (*count)++;

Part D: Using an editor on student.cs.uni.edu open the file averageScores2.c which also interactively averages a collection of scores, but this program:
- asks the user to enter the number of scores first, then
- dynamically allocates an array just big enough to hold the scores using malloc (memory allocate).
Recall that malloc takes as a parameter the size of the array in bytes and return a pointer to the first element of the dynamically allocated array in the heap. Typically, you use the sizeof function to determine the size of a single element and multiply by the number of elements to calculate the size of the array in bytes. The malloc function returns “generic” void * pointer type which must be cast to a pointer of the appropriate type.
Answer the following questions about the `averageScores2.c` program:

a) When the main program starts execution where does the `scores` pointer point?

The main calls `getScores` as:

```c
getScores(&numberOfScores, &scores);
```

b) What is the purpose of `&` symbol in front of the `scores` parameter?

The `getScores` function definition is:

```c
void getScores(int * count, double ** scores) {
    double score;
    double * localScoresPtr;
    int i;

    printf("Enter the # of scores you will be entering: ");
    scanf("%d", count);
    localScoresPtr = (double *) malloc(sizeof(double)*(*count));

    for (i = 0; i < *count; i++) {
        printf("Enter a score: ");
        scanf("%lf", &score);
        localScoresPtr[i] = score;
    } // end for

    *scores = localScoresPtr;
} // end getScores
```

c) What is the purpose of `**` symbols in front of the `scores` parameter in the function header?

d) Explain all parts of the assignment statement:

```c
localScoresPtr = (double *) malloc(sizeof(double)*(*count));
```

e) What type of value is assignment by the assignment statement?

```
*scores = localScoresPtr;
```

**Part E:** Complete the program called `displayMultiplicationTable.c` that asks the user to enter two integers: `value1` and `value2`, then prints a multiplication table with:

- rows labeled 1 to `value1`,
- columns labeled 1 to `value2`, and
- each value in the table being the product of the corresponding row and column label

The main function acts as a high-level outline for the program and calls three functions:

- `getValues` - allows the user to enter and returns `value1` and `value2` values
- `calculateRowProducts` - passed a row value, the `value2`, and returns an array of products from (row x 1) to (row x value2)
- `printTableHeading` - passed the `value1` and `value2` values and prints the table header
- `printRow` - passed a row value, the `value2`, and corresponding `rowProducts` array which is printed

Submit `lab2.zip` containing question answers and completed programs on eLearning system.
Learning Objectives:
- Contrast and compare the three types of cache: direct-mapped, fully associative, and set-associative
- Compute the page-table given a diagram of main memory frames, and translate from a logical/virtual address to a physical address.
- Design efficient programs that utilize the cache optimally.

1. Suppose we have a 4 GB \(2^{32}\) bytes) memory that is byte addressable, and a 64KB \(2^{16}\) bytes) cache with 32 \(2^{5}\) bytes per block.
   a) How many total lines are in the cache?
   
b) If the cache is direct-mapped, how many cache lines could a specific memory block be mapped to?
   
c) If the cache is direct-mapped, what would be the format (tag bits, cache line bits, block offset bits) of the address? (Clearly indicate the number of bits in each)
   
d) If the cache is fully-associative, how many cache lines could a specific memory block be mapped to?
   
e) If the cache is fully-associative, what would be the format of the address?
   
f) If the cache is 4-way set associative, how many cache lines could a specific memory block be mapped to?
   
g) If the cache is 4-way set associative, how many sets would there be?
   
h) If the cache is 4-way set associative, what would be the format of the address?

2. Consider the following two sections of C code that both sum the elements of a 10,000 x 10,000 two-dimensional array M which contains floating points.

<table>
<thead>
<tr>
<th>Code A</th>
<th>Code B</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum = 0.0;</td>
<td>sum = 0.0;</td>
</tr>
<tr>
<td>for (r = 0; r &lt; 10000; r++)</td>
<td>for (c = 0; c &lt; 10000; c++)</td>
</tr>
<tr>
<td>for (c = 0; c &lt; 10000; c++)</td>
<td>for (r = 0; r &lt; 10000; r++)</td>
</tr>
<tr>
<td>sum = sum + M[r][c];</td>
<td>sum = sum + M[r][c];</td>
</tr>
</tbody>
</table>

Explain why Code A takes 1.27 seconds while Code B takes 2.89 seconds. Hint: C uses row-major ordering to store two-dimensional arrays i.e.,

![Diagram of a two-dimensional array M with row and column indices and corresponding values]
3. Consider the demand paging system with 4096-byte pages.

<table>
<thead>
<tr>
<th>Page Table for B</th>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid Bit</td>
<td>Frame Number</td>
</tr>
<tr>
<td>Frame#</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>page 2 of A</td>
</tr>
<tr>
<td>1</td>
<td>page 4 of A</td>
</tr>
<tr>
<td>2</td>
<td>page 2 of B</td>
</tr>
<tr>
<td>3</td>
<td>page 1 of B</td>
</tr>
<tr>
<td>4</td>
<td>page 1 of A</td>
</tr>
<tr>
<td>5</td>
<td>page 4 of B</td>
</tr>
<tr>
<td>6</td>
<td>page 6 of A</td>
</tr>
</tbody>
</table>

- CPU
- Process A
  - page 0
  - page 1
  - page 2
  - page 3
  - page 4
  - page 5
  - page 6

(a) Complete the above page table for Process B.

(b) If process B is currently running and the CPU generates a logical/virtual address of $5010_{10}$, then what would be the corresponding physical address?

4. Explain how a TLB (translation-lookaside buffer) speeds the process of address translation?

5. (This question deals with the following toy virtual memory system on the next page which is tiny...)

You have a byte-addressable memory with 8 bytes per memory block. The memory management unit has a two-entry TLB (fully-associate cache with a Page # as the tag) and a slower (vague I know) page-table for a process P. The cache is 2-way set-associative and has a total of 4 cache lines (tag bits shown in binary). Assume page size of 16 bytes, so two memory blocks per frame. In the diagram, memory is divided into blocks, where each block’s content is represented abstractly by a letter.

Given the system state as depicted above, answer the following questions:

(a) How many bits are in a virtual address for process P?

(b) How many bits are in a physical address?

(c) Show the address format for a logical/virtual address including field names and number of bits.

(d) Using your format in part (c), convert the virtual address $50_{10}$ to binary and put it in the appropriate fields. Now, explain how these fields are used to translate to the corresponding physical address.

(e) Show the address format for a physical address including field names and number of bits that are used to check the cache.

(f) Given that virtual address $12_{10}$ translates to physical address $60_{10}$. Using your format in part (c), convert the physical address $60_{10}$ to binary and put it in the appropriate fields. Now, explain how these fields are used to locate physical address 60 in the cache.

(g) Given that virtual address $100_{10}$ is located on virtual page 6, offset 4. Indicate exactly how this address would be translated to its corresponding physical address and how the data would be accessed. Include in your explanation how the TLB, the page table, cache, and memory are used.
## Operator Precedence and Associativity

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
<th>Usage(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>::</td>
<td>unary: left-to-right binary: right-to-left</td>
<td></td>
</tr>
<tr>
<td>( ) [ ] .</td>
<td>left-to-right</td>
<td>parenthesis index object pointer/structure pointer dot operator</td>
</tr>
<tr>
<td>++ -- - + ! ~ (type) * &amp; sizeof</td>
<td>right-to-left</td>
<td>increment and decrement unary negation and plus logical negation one's complement operator type cast indirect address-of/reference</td>
</tr>
<tr>
<td>* / %</td>
<td>left-to-right</td>
<td>multiply, division, remainder</td>
</tr>
<tr>
<td>+ -</td>
<td>left-to-right</td>
<td>addition and subtraction</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left-to-right</td>
<td>io: insertion and extraction bit-wise shift left and right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left-to-right</td>
<td>comparisons for inequality</td>
</tr>
<tr>
<td>== !=</td>
<td>left-to-right</td>
<td>comparison for equality</td>
</tr>
<tr>
<td>&amp; ^</td>
<td>left-to-right</td>
<td>bit-wise AND bit-wise exclusive-OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>left-to-right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>? :</td>
<td>right-to-left</td>
<td>conditional</td>
</tr>
<tr>
<td>= += -= *= /= %= &amp;= ^=</td>
<td>right-to-left</td>
<td>assignment</td>
</tr>
<tr>
<td>,</td>
<td>left-to-right</td>
<td>comma operator</td>
</tr>
</tbody>
</table>
C function calls:

Set up a call frame on run-time stack for function

int x = 1;
main (double y = 3.4);
A (x, y); (*);

"actual parameters"

Call-frame contains
- Return address - location
  in code where function was called
- Parameters passed
  C only has pass-by-value
- Local variables "formal parameters"

void A (int pl, double p2) {
  int i;
  printf ("Param 1: %d, Param 2: %f\n", pl, p2);

return from function or parameter
causes the call-frame to be
popped from run-time stack
displayRectInfo.c

/* Program to demonstrate functions and parameter passing.
   Compile with math library: gcc -o rect displayRectInfo.c
   Run by: ./rect
*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

// function prototypes
void getDimensions(double * length, double * width);
double calculateArea(double length, double width);
void calculatePerimeter(double, double, double *);
void displayRectInformation(double, double, double, double, double);

int main() {
    double length, width, area, perimeter;

    getDimensions(&length, &width);

    area = calculateArea(length, width);

    calculatePerimeter(length, width, &perimeter);

    displayRectInformation(length, width, area, perimeter);
}

/***********************
* Procedure getDimensions asks the user to enter the length and width of the rectangle and returns them.
* void getDimensions(double * length, double * width) {

    printf("Enter the length of a rectangle: ");
    scanf("%lf", length); // NOTE %lf for double, but %f for float

    printf("Enter the width of a rectangle: ");
    scanf("%lf", width); // NOTE %lf for double, but %f for float

}

/***********************
* Function calculateArea is passed the dimensions of the rectangle and returns its area.
* double calculateArea(double length, double width) {

    area = length * width;
    return area;
}

***********************
* Procedure calculatePerimeter is passed the length and width of
void calculatePerimeter(double length, double width, double *perimeter) {
    *perimeter = 2.0 * (length + width);
} // end calculatePerimeter

void displayRectInformation(double length, double width, double area, double perimeter) {
    printf("A rectangle with a length of %.2f and a width of %.2f has an area of %.2f\n", length, width, area);
    printf("and a perimeter of %.2f.\n", perimeter);
} // end displayRectInformation
```plaintext
function tripleInteger

int tripleInteger (int value)

return value * 3;

main:

int x = 1;
int result;

result = tripleInteger(x);
```

Diagram:

Return value

3 call-frame

3 tripleInteger

3 main

3 tripleInteger(x)

x

x3

L2-4
arrays in C - collection of same type values:

main:
    double scores[100]; // static allocation

Scores:

\[
\begin{array}{c|c|c|c|c}
0 & 1 & 2 & 3 & 99 \\
\hline
\frac{50}{10} & 51.5 & 52 & 99
\end{array}
\]

\[
\text{scores}[2] = 50.5
\]

\[
\text{scores}[2] = \text{scores}[2] + 1
\]

main:
    double scores[100];
    int length;
    
    getScores(\&length, \&scores);
void getScores (int * length, double scores[10])
{
    printf("How many scores?\n");
    scanf("%d", length);
    for (index = 0; index < *length; index++)
    {
        scanf("%lf", &scores[index]);
        scanf("%lf", scores + index);
    }
}

// end getScores

point to m allocate (void)

malloc ( )

# bytes
main:

double * scores,
int length;

getScoresDynamic(&length, &scores);

void getScoresDynamic(int * length, double ** scores);

int index;

printf("How many scores?: ");
scanf("%d", length);

*score = (double*)malloc(sizeof(double) * (*length))

for

L2-4
Memory Hierarchy Supplement

Goal: "Fast" "unlimited" storage at a reasonable cost per bit.

Recall the von Neumann bottleneck - single, relatively slow path between the CPU and main memory.

Fast: When you need something from "memory" check "faster" cache(s) first for a copy

"Unlimited" storage: Virtual memory - executing program's logical address space (ML pgm, run-time stack, heap, global memory) completing out on disk with main memory (DRAM) acting like "cache" for hard disk.
Main Idea of a Cache - keep a copy of frequently used information as "close" (w.r.t access time) to the processor as possible.

Steps when the CPU generates a memory request:
1) check the (faster) cache first
2) If the addressed memory value is in the cache (called a hit), then no need to access memory
3) If the addressed memory value is NOT in the cache (called a miss), then transfer the block of memory containing the reference to cache. (The CPU is stalled and idle while this occurs)
4) The cache supplies the memory value from the cache.

Effective (Average) Memory Access Time
Suppose that the hit time (i.e., access time of cache, \( t_h \)) is 2 ns, the cache miss penalty (i.e., load cache line from memory might involve multiple reads) is 150 ns, and the hit ratio is 99% (so miss ratio is 1%).

\[
\text{Effective Access Time} \approx (\text{hit time}) + (\text{miss penalty} \times \text{miss ratio})
\]

\[
\text{Effective Access Time} = 2 + 150 \times (1 - 0.99) = 2 + 1.5 = 3.5 \text{ ns}
\]

(One way to reduce the miss penalty is to not have the cache wait for the whole block to be read from memory before supplying the accessed memory word.)
Fortunately, programs exhibit *locality of reference* that helps achieve high hit-ratios:

1) *spatial locality* - if a (logical) memory address is referenced, nearby memory addresses will tend to be referenced soon.

2) *temporal locality* - if a memory address is referenced, it will tend to be referenced again soon.
Three Types of Cache

Cache - Small fast memory between the CPU and RAM/Main memory.

Example:
- 32-bit logical/virtual address
- 512 KB ($2^{19}$) cache size (assume only one level of cache)
- 8 byte per block/line
- byte-addressable memory

Number of Cache Line = \( \frac{\text{size of cache}}{\text{size of line}} = \frac{2^{19}}{2^3} = 2^{16} \).

1) Direct-mapped - a memory block maps to a single cache line
Same Cache Example:
- 32-bit logical/virtual address, byte-addressable memory
- 512 KB \( (2^{19}) \) cache size (assume only one level of cache)
- 8 byte per block/line

Number of Cache Line = \( \frac{\text{size of cache}}{\text{size of line}} = \frac{2^{19}}{2^3} = 2^{16} \)

2) Fully-Associative Cache - a memory block can map to any cache line

Advantage: Flexibility on what's in the cache
Disadvantage: Complex circuit to compare all tags of the cache with the tag in the target address
Therefore, they are expensive and slower so use only for small caches (say 8-64 lines)

Replacement algorithms - on a miss of a full cache, we must select a block in the cache to replace
- LRU - replace the cache block that has not been used for the longest time (need additional usage bits for each line)
- Random - select a block randomly (only slightly worse that LRU and easier to implement)
Same Cache Example:
- 32-bit logical/virtual address, byte-addressable memory
- 512 KB ($2^{19}$) cache size (assume only one level of cache)
- 8 byte per block/line

\[
\text{Number of Cache Line} = \frac{\text{size of cache}}{\text{size of line}} = \frac{2^{19}}{2^3} = 2^{16}
\]

Set-Associative Cache - a memory block can map to a small (2, 4, or 8) set of cache lines

Common Possibilities:
- 2-way set associative - each memory block can map to either of two lines in the cache.
- 4-way set associative - each memory block can map to either of four lines in the cache

\[
\text{Number of Sets} = \frac{\text{number of cache lines}}{\text{size of each set}} = \frac{2^{16}}{4} = \frac{2^{16}}{2^2} = 2^{14}
\]

4-way Set Associative Cache

Memory

<table>
<thead>
<tr>
<th>Other Bits</th>
<th>way 0</th>
<th>way 1</th>
<th>way 2</th>
<th>way 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>0</td>
<td>2^{14}</td>
<td>2^1</td>
<td>2^{14}</td>
</tr>
<tr>
<td></td>
<td>_tag</td>
<td>_tag</td>
<td>_tag</td>
<td>_tag</td>
</tr>
<tr>
<td></td>
<td>block</td>
<td>block</td>
<td>block</td>
<td>block</td>
</tr>
</tbody>
</table>

CPU

32-bit logical/virtual address:

```
15 14 3

\_tag \_set \_offset
```
Typical system view of the memory hierarchy

Virtual Memory - programmer views memory as large address space without concerns about the amount of physical memory or memory management. (What do the terms 32-bit (or 64-bit) operating system mean?)

Benefits:
1) programs can be bigger that physical memory size since only a portion of them may actually be in physical memory
2) higher degree of multiprogramming is possible since only portions of programs are in memory

An Operating System goal with hardware support is to make virtual memory efficient and transparent to the user.

Memory-Management Unit (MMU) for paging

Note: The "Valid" bit is sometimes called the Resident R-bit.
Demand paging is a common way for OSs to implement virtual memory. Demand paging ("lazy pager") only brings a page into physical memory when it is needed. A "Valid bit" is used in a page table entry to indicate if the page is in memory or only on disk.

A page fault occurs when the CPU generates a logical address for a page that is not in physical memory. The MMU will cause a page-fault trap (interrupt) to the OS.

Steps for OS's page-fault trap handler:
1) Check page table to see if the page is exists in logical address space. If it is invalid, terminate the process; otherwise continue.
2) Find a free frame in physical memory (take one from the free-frame list or replace a page currently in memory).
3) Schedule a disk read operation to bring the page into the free page frame. (We might first need to schedule a previous disk write operation to update the virtual memory copy of a "dirty" page that we are replacing.)
4) Since the disk operations are sooooo slowooooow, the OS would context switch to another ready process selected from the ready queue.
5) After the disk (a DMA device) reads the page into memory, it invokes an I/O completion interrupt. The OS will then update the PCB and page table for the process to indicate that the page is now in memory and the process is ready to run.
6) When the process is selected by the short-term scheduler to run, it repeats the instruction that caused the page fault. The memory reference that caused the page fault will now succeed.

Performance of Demand Paging
To achieve acceptable performance degradation (5-10%) of our virtual memory, we must have a very low page fault rate (probability that a page fault will occur on a memory reference).

When does a CPU perform a memory reference?
1) Fetch instructions into CPU to be executed
2) Fetch operands used in an instruction (load and store instructions on RISC machines)

Example:
Let \( p \) be the page fault rate, and \( ma \) be the memory-access time.
Assume that \( p = 0.02 \), \( ma = 50 \) ns and the time to perform a page fault is 12,200,000 ns (12.2 ms).

\[
\frac{\text{effective memory access time}}{\text{access time}} = \left( \frac{\text{prob. of no page fault}}{\text{main memory access time}} \right) + \left( \frac{\text{prob. of page fault}}{\text{page fault time}} \right)
\]

\[
= (1 - p) \times 50\text{ns} + p \times 12,200,000
\]

\[
= 0.98 \times 50\text{ns} + 0.02 \times 12,200,000
\]

\[
= 244,049\text{ns}
\]

The program would appear to run very slowly!!!

If we only want say 10% slow down of our memory, then the page fault rate must be much better!

\[
55 = (1 - p) \times 50\text{ns} + p \times 12,200,000\text{ns}
\]

\[
55 = 50 - 50p + 12,200,000p
\]

\[
p = 0.0000004\text{, or 1 page fault in 2,439,990 references}
\]

Fortunately, programs exhibit locality of reference that helps achieve low page-fault rates. Page size is typically 4 KB.
Storage of the Page Table Issues
1) Where is it located?
If it is in memory, then each memory reference in the program results in two memory accesses: one for the page table entry, and another to perform the desired memory access.

Solution: TLB (Translation-lookaside Buffer) - small, fully-associative cache to hold PT entries
Ideally, when the CPU generates a memory reference, the PT entry is found in the TLB, the page is in memory, and the block with the page is in the cache, so no memory accesses are needed.
However, each CPU memory reference involves two cache lookups and these cache lookups must be done sequentially, i.e., first check TLB to get physical frame # used to build the physical address, then use the physical address to check the tag of the L1 cache.

Alternatively, the L1 cache can contain virtual addresses (called a virtual cache). This allows the TLB and cache access to be done in parallel. If the cache hits, the result of the TLB is not used. If the cache misses, then the address translation is under way and used by the L2 cache.

2) Ways to handle large page tables:
Page table for each process can be large
E.g., 32-bit address, 4 KB (2^12 bytes) pages, byte-addressable memory, 4 byte PT entry

1 M (2^20) of page table entries, or 4 MB for the whole page table with 4 byte page table entries

A solution:
a) two-level page table - the first level (the "directory") acts as an index into the page table which is scattered across several pages. Consider a 32-bit example with 4KB pages and 4 byte page table entries.

Problem with paging:
1) Protection unit is a page, i.e., each Page Table Entry can contain protection information, but the virtual address space is divided into pages along arbitrary 4KB boundaries.
Segmentation - divides virtual address space in terms of meaningful program modules which allows each to be associated with different protection. For example, a segment containing a matrix multiplication subprogram could be shared by several programs.

Programmer views memory as multiple address spaces, i.e., segments. Memory references consist of two parts: <segment #, offset within segment>.

As in paging, the operating system with hardware support can move segments into and out of memory as needed by the program.

Each process (running program) has its own segment table similar to a page table for performing address translations.

Problems with Segmentation:
1) hard to manage memory efficiently due to external fragmentation
2) segments can be large in size so not many can be loaded into memory at one time

Solution: Combination of paging with segmentation by paging each segment.
Week 3 Discussion Questions
Chapter 2.2.5 - 2.2.6:

1. Assume that an automobile assembly process takes 4 hours.

   Chassis  | Motor  | Interior  | Exterior
   ![Car Chassis](image) | ![Car Motor](image) | ![Car Interior](image) | ![Car Exterior](image)

   a) If the stages take the following amounts of time, then what is the time between completions of automobiles?
   - Chassis 45 minutes
   - Motor 1 hour
   - Interior 1 hour and 15 minutes
   - Exterior 1 hour

2. Two approaches for designing a computer is CISC (Complex Instr. Set Computer - pre-1980) and RISC (Reduced Instruction Set Computer post 1985, MIPS was one of the first commercial RISC processor). A CISC philosophy was to make assembly language (AL) as much like a high-level language (HLL) as possible to reduce the "semantic gap" between AL and HLL. The rational for CISC at the time was to:

   • reduce compiler complexity and aid assembly language programming. Compilers were not too good during the 50's to 70's, (e.g., they made poor use of general purpose registers so code was inefficient) so some programs were written in assembly language.
   • reduce the program size. More powerful/complex instructions reduced the number of instructions necessary in a program. Memory during the 50's to 70's was limited and expensive.
   • improve code efficiency by allowing complex sequence of instructions to be implemented in microcode. For example, the Digital Equipment Corporation (DEC) VAX computer had an assembly-language instruction "MATCHC substrLength, substr, strLength, str" that looks for a substring within a string.

   The architectural characteristics of CISC machines include:
   • complex, high-level like AL instructions
   • variable format machine-language instructions that execute using a variable number of clock cycles
   • many addressing modes (e.g., the DEC VAX had 22 addressing modes)

   a) Why are complex instructions of CISC (Complex Instr. Set Computer) machines difficult to pipeline?

   b) Why are RISC machines usually Load & Store machines (i.e., only Load and Store instructions access memory)?

3. Intel x86 architecture (e.g., Pentium 4, Xeon, Core 2 Duo, i3, i5, i7, AMD Athon, ... processors) has been around since the '70s so it is a CISC instruction set. To take advantage of later pipelining and superscalar concepts. The Pentium 4 for example operates by:

   • Fetching x86 (CISC) instructions from memory in order the of static program
   • Translating each x86 instruction into one or more fixed length RISC instructions (micro-operations)
   • Execute micro-ops on superscalar pipeline where
     - micro-ops may be executed out of order
     - up to 4 micro-ops dispatched per clock cycle
   • Commit results of micro-ops to register set in original x86 program flow order

   Why didn't Intel scrap the x86 architecture and design a new RISC and superscalar architecture?
Learning Objectives:
- Write correct programs using command-line arguments.
- Write correct programs that use the C string commands.
- Write correct programs that use array indexing and pointer arithmetic to access array elements.

To start the lab:
- watch the Lab 3 Video on the eLearning system
- download lab3.zip from the eLearning system and unzip/extract it locally on your computer
- copy the lab3 directory to student.cs.uni.edu using a secure ftp client (winSCP, FileZilla, scp, etc.)
- log-on to student.cs.uni.edu using Putty/ssh

Part A: Using an editor on student.cs.uni.edu open the file ageCmdLine.c which contains a simple C program that allows the user to enter their name and age on the command-line, and it echo's back both to the user.

Answer the following questions about the ageCmdLine.c program when it is:
Compile by: gcc -o age ageCmdLine.c
Run by: ./age Bob 13
Output: Bob your age is 13.

a) What is the purpose of adding 1 to the string length of the name in the function call:
   name = (char *) malloc(sizeof(char)*(strlen(argv[1])+1));

b) How does the strcpy function know when the whole string has been copied?
   strcpy(name, argv[1]);

c) Why does the age parameter have a & symbol in front of it in the sscanf call?
   sscanf(argv[2], "%d", &age);

d) Why is it generally important to deallocate dynamically-allocated (i.e., created using malloc) arrays using the free function (i.e., free(name));?
Part B: Write a similar program called makeReverseString.c that uses the command-line arguments and the string.h functions to concatenate all, but the first command-line argument in reverse order into a single string with a space-character between each. For example, running the program ./reverseString with command-line of:
./reverseString Today is a good day to learn C
creates a single string of: C learn to day good a is Today

Part C: Using an editor on student.cs.uni.edu open the file calcAverage.c which contains a simple C program to interactively allow the user to average a collection of numbers on command-line.
Answer the following questions about the calcAverage.c program:
a) What is the maximum number of values that can be handled by this program?

b) What is the purpose of if-statement (i.e., if(argc == 1)...)?

c) The for-loop in the main program uses scanf to convert a command-line argument from argv array to a double and stores it in the values array. Why is the & symbol needed in front of the values[index] parameter?

d) The for-loop in the main program also contains two alternative scanf options to illustrate the usage of pointer arithmetic to address/access array elements. In the middle option, why is the & symbol not needed in front of the values+index parameter?

e) To practice pointer arithmetic, modify the updating of the total (i.e., total = total + scores[i];) in the calculateAverage function, so that it uses pointer arithmetic instead of indexing. Re-compile and re-run the program to make sure that it still works.

Submit lab3.zip containing question answers and completed programs on eLearning system
Learning Objectives:
- Produce the timing diagram for a 5-stage pipeline with and without forwarding/by-pass-signal paths.
- Describe how PC-relative addressing is used in conditional branch instructions to reference a label (e.g., ELSE:) in memory, and how base register-offset address can be used to access local variables within a function's call-frame and to access global variables.
- Explain the operation of a branch-predication buffer (BPB/BHT) and how it reduces the number of branch penalties.
- Identify the location of conditional and unconditional branch instructions within a HLL program, and compute the number of branch penalties for each conditional and unconditional branch in a 5-stage pipeline utilizing static branch predictions, 1-bit branch-prediction buffer, and 2-bit branch-prediction buffer.
- Summarize the characteristics of a superscalar processor.
- Identify the WAR (write-after-read) and WAW (write-after-write) dependencies within a section of code.
- Explain how register renaming eliminates WAR and WAW to increase instruction-level parallelize.

1. The whole question refers to the pipelined, RISC machine with five stages:
   - F, fetch - fetch the instruction from memory
   - D, decode - determine the type of instruction and read any necessary register values
   - E, execute - perform ALU operation or memory address calculation for LOAD or STORE instructions
   - M, memory - access memory on LOAD or STORE instruction
   - W, write - write register values

a) Complete the following timing diagram assuming NO by-pass signal paths.

<table>
<thead>
<tr>
<th>Without by-pass signal paths</th>
<th>Time →</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>ADD R3, R2, R1</td>
<td>F</td>
</tr>
<tr>
<td>LOAD R4, 16(R3)</td>
<td></td>
</tr>
<tr>
<td>STORE R1, 8(R4)</td>
<td></td>
</tr>
<tr>
<td>SUB R3, R4, R1</td>
<td></td>
</tr>
<tr>
<td>MUL R6, R3, R4</td>
<td></td>
</tr>
<tr>
<td>STORE R6, 4(R5)</td>
<td></td>
</tr>
</tbody>
</table>

b) Complete the following timing diagram assuming by-pass signal paths.

<table>
<thead>
<tr>
<th>With by-pass signal paths</th>
<th>Time →</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>ADD R3, R2, R1</td>
<td>F</td>
</tr>
<tr>
<td>LOAD R4, 16(R3)</td>
<td></td>
</tr>
<tr>
<td>STORE R1, 8(R4)</td>
<td></td>
</tr>
<tr>
<td>SUB R3, R4, R1</td>
<td></td>
</tr>
<tr>
<td>MUL R6, R3, R4</td>
<td></td>
</tr>
<tr>
<td>STORE R6, 4(R5)</td>
<td></td>
</tr>
</tbody>
</table>
c. Draw ALL the bypass-signal paths and MUXs needed for the above example.

2. Another simple sort is called insertion sort. Recall that in a simple sort:
   - the outer loop keeps track of the dividing line between the sorted and unsorted part with the sorted part growing by one in size each iteration of the outer loop. (below the firstUnsortedIndex keeps track of the dividing line)
   - the inner loop's job is to do the work of extending the sorted part's size by one.

After several iterations of insertion sort's outer loop, an array might look like:

<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unssorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5</td>
<td>6 7 8</td>
</tr>
<tr>
<td>10 20 35 40</td>
<td>45 60</td>
</tr>
<tr>
<td>(25)</td>
<td>50 90</td>
</tr>
<tr>
<td>* * *</td>
<td>* * *</td>
</tr>
</tbody>
</table>

In insertion sort the inner-loop takes the "first unsorted item" (25 at index 6 in the above example) and "inserts" it into the sorted part of the array "at the correct spot." After 25 is inserted into the sorted part, the array would look like:

<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unssorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5</td>
<td>6 7 8</td>
</tr>
<tr>
<td>10 20 25 35</td>
<td>40 45 60</td>
</tr>
<tr>
<td>50 90</td>
<td>* * *</td>
</tr>
</tbody>
</table>

Consider the following insertion sort algorithm that sorts an array numbers:

InsertionSort(numbers - address to integer array, length - integer)

integer firstUnsortedIndex, testIndex, elementToInsert;

for firstUnsortedIndex = 1 to (length-1) do
    testIndex = firstUnsortedIndex-1;
    elementToInsert = numbers[firstUnsortedIndex];
    while (testIndex >=0) AND (numbers[testIndex] > elementToInsert ) do
        numbers[ testIndex + 1 ] = numbers[ testIndex ];
        testIndex = testIndex - 1;
    end while
    numbers[ testIndex + 1 ] = elementToInsert;
end for
end InsertionSort

a) Where in the above code would unconditional branches be used and where would conditional branches be used?
Homework #3

b) Assumptions:
• length = 100 and the numbers are initially in descending order before the insertion sort algorithm is called
• the five-stage pipeline discussed in class
• the outcome of conditional branches is known at the end of the E stage
• target addresses of all branches is known at the end of the D stage
• ignore any data hazards
Under the above assumptions, answer the following questions:
i) If NO branch-prediction buffer is used (i.e., hardware continues to fetch sequential until the outcome of the branch is determined), then what will be the total branch penalty (# cycles wasted) for the algorithm?

ii) If a branch-prediction buffer with one history bit per entry is used, then what will be the total branch penalty (# cycles wasted) for the algorithm? (Assume predict-not taken is used if there is no match in the branch-prediction buffer) Explain your answer.

iii) If a branch-prediction buffer with two history bit per entry is used, then what will be the total branch penalty (# cycles wasted) for the algorithm? (Assume predict-not taken is used if there is no match in the branch-prediction buffer) Explain your answer.

3. Consider the following sequence of assembly language program:
Instruction 1: MUL R3, R4, R5
Instruction 2: ADD R4, R5, R7
Instruction 3: SUB R6, R4, R3
Instruction 4: ADD R3, R2, R1
a) Identify all RAW dependencies in the above code (formatted as “Instructions # and # on register R#”)

b) Identify all WAW dependencies in the above code (formatted as “Instructions # and # on register R#”)

c) Identify all WAR dependencies in the above code (formatted as “Instructions # and # on register R#”)

d) Assume that the “programmer” visible architecture had registers R0 to R31, but the superscalar processor utilized dynamic register renaming with registers R33 - R64. Rewrite the above instructions utilizing register renaming to eliminate the WAR and WAW dependencies?

e) Explain how your register-renamed code in (d) has more instruction-level parallelism (ILP).
1. Assume that an automobile assembly process takes 4 hours.

   Chassis 1 hour  
   Motor 1 hour  
   Interior 1 hour  
   Exterior 1 hour

a) If the stages take the following amounts of time, then what is the time between completions of automobiles?  
   total time  
   # stages

b) What if the stages were not divided equally?

c) What if the amount of time at each stage varied depending on the options (e.g., deluxe interior package, towing package)?

2. We could follow the instruction/machine cycle into stages for instruction pipelining.

   Fetch Instruction - read instruction pointed at by the program counter (PC) from memory into Instr. Reg. (IR)
   Decode Instruction - figure out what kind of instruction was read
   Fetch Operands - get operand values from the memory or registers
   Execute Instruction - do some operation with the operands to get some result
   Write Result - put the result into a register or in a memory location

Two approaches for designing a computer are CISC (Complex Instr. Set Computer - pre-1980) and RISC (Reduced Instruction Set Computer post 1985, MIPS was one of the first commercial RISC processor). A CISC philosophy was to make assembly language (AL) as much like a high-level language (HLL) as possible to reduce the "semantic gap" between AL and HLL. The rational for CISC at the time was to:

- reduce computer complexity and aid assembly language programming. Compilers were not too good during the 50's to 70's, (e.g., they made poor use of general purpose registers so code was inefficient) so some programs were written in assembly language.
- reduce the program size. More powerful/complex instructions reduced the number of instructions necessary in a program. Memory during the 50's to 70's was limited and expensive.
- improve code efficiency by allowing complex sequence of instructions to be implemented in microcode. For example, the Digital Equipment Corporation (DEC) VAX computer had an assembly-language instruction "MATCHC substrLength, substr, strLength, str" that looks for a substring within a string.

The architectural characteristics of CISC machines include:
- complex, high-level like AL instructions
- variable format machine-language instructions that execute using a variable number of clock cycles
- many addressing modes (e.g., the DEC VAX had 22 addressing modes)

   a) What are the architectural characteristics of RISC machines?

- fixed length instr. (ML) 32-bits only
- only reg. operands for calculations
- load/store machine
- large # of registers
- 32-bits, Memory 7-bits
- add R4, R5, R6
- add R5, R6
"The big blue horse."

"blue"

Memory

CPU

栈

Global Data

.text

.data

2D bit

load R5

Load R5

load R5

Effective addr. (EA) = (base reg) + offset
3. The whole question refers to a pipelined, RISC machine with five stages:
   - F, fetch - fetch the instruction from memory
   - D, decode - determine the type of instruction and read any necessary register values
   - E, execute - perform ALU operation or memory address calculation for LOAD or STORE instructions
   - M, memory - access memory on LOAD or STORE instruction
   - W, write - write register values

a) Complete the following timing diagram assuming no by-pass signal paths:

<table>
<thead>
<tr>
<th>Without by-pass signal paths</th>
<th>Time →</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1, R3, R4</td>
<td>F</td>
</tr>
<tr>
<td>ADD R2, R4, R5</td>
<td>F</td>
</tr>
<tr>
<td>ADD R3, R2, R1</td>
<td>F</td>
</tr>
<tr>
<td>LOAD (R2), J12(R3)</td>
<td>F</td>
</tr>
<tr>
<td>STORE (R2), 16(R2)</td>
<td>F</td>
</tr>
<tr>
<td>ADD R1, R3, R4</td>
<td>D</td>
</tr>
<tr>
<td>ADD R2, R4, R5</td>
<td>D</td>
</tr>
<tr>
<td>ADD R3, R2, R1</td>
<td>D</td>
</tr>
<tr>
<td>LOAD (R2), J12(R3)</td>
<td>D</td>
</tr>
<tr>
<td>STORE (R2), 16(R2)</td>
<td>D</td>
</tr>
<tr>
<td>ADD R1, R3, R4</td>
<td>E</td>
</tr>
<tr>
<td>ADD R2, R4, R5</td>
<td>E</td>
</tr>
<tr>
<td>ADD R3, R2, R1</td>
<td>E</td>
</tr>
<tr>
<td>LOAD (R2), J12(R3)</td>
<td>E</td>
</tr>
<tr>
<td>STORE (R2), 16(R2)</td>
<td>E</td>
</tr>
<tr>
<td>ADD R1, R3, R4</td>
<td>M</td>
</tr>
<tr>
<td>ADD R2, R4, R5</td>
<td>M</td>
</tr>
<tr>
<td>ADD R3, R2, R1</td>
<td>M</td>
</tr>
<tr>
<td>LOAD (R2), J12(R3)</td>
<td>M</td>
</tr>
<tr>
<td>STORE (R2), 16(R2)</td>
<td>M</td>
</tr>
<tr>
<td>ADD R1, R3, R4</td>
<td>W</td>
</tr>
<tr>
<td>ADD R2, R4, R5</td>
<td>W</td>
</tr>
<tr>
<td>ADD R3, R2, R1</td>
<td>W</td>
</tr>
<tr>
<td>LOAD (R2), J12(R3)</td>
<td>W</td>
</tr>
<tr>
<td>STORE (R2), 16(R2)</td>
<td>W</td>
</tr>
</tbody>
</table>

b) Complete the following timing diagram assuming by-pass signal paths as shown above:

<table>
<thead>
<tr>
<th>With by-pass signal paths</th>
<th>Time →</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1, R3, R4</td>
<td>F</td>
</tr>
<tr>
<td>ADD R2, R4, R5</td>
<td>F</td>
</tr>
<tr>
<td>ADD R3, R2, R1</td>
<td>F</td>
</tr>
<tr>
<td>LOAD (R2), J12(R3)</td>
<td>F</td>
</tr>
<tr>
<td>STORE (R2), 16(R2)</td>
<td>F</td>
</tr>
<tr>
<td>ADD R1, R3, R4</td>
<td>D</td>
</tr>
<tr>
<td>ADD R2, R4, R5</td>
<td>D</td>
</tr>
<tr>
<td>ADD R3, R2, R1</td>
<td>D</td>
</tr>
<tr>
<td>LOAD (R2), J12(R3)</td>
<td>D</td>
</tr>
<tr>
<td>STORE (R2), 16(R2)</td>
<td>D</td>
</tr>
<tr>
<td>ADD R1, R3, R4</td>
<td>E</td>
</tr>
<tr>
<td>ADD R2, R4, R5</td>
<td>E</td>
</tr>
<tr>
<td>ADD R3, R2, R1</td>
<td>E</td>
</tr>
<tr>
<td>LOAD (R2), J12(R3)</td>
<td>E</td>
</tr>
<tr>
<td>STORE (R2), 16(R2)</td>
<td>E</td>
</tr>
<tr>
<td>ADD R1, R3, R4</td>
<td>M</td>
</tr>
<tr>
<td>ADD R2, R4, R5</td>
<td>M</td>
</tr>
<tr>
<td>ADD R3, R2, R1</td>
<td>M</td>
</tr>
<tr>
<td>LOAD (R2), J12(R3)</td>
<td>M</td>
</tr>
<tr>
<td>STORE (R2), 16(R2)</td>
<td>M</td>
</tr>
<tr>
<td>ADD R1, R3, R4</td>
<td>W</td>
</tr>
<tr>
<td>ADD R2, R4, R5</td>
<td>W</td>
</tr>
<tr>
<td>ADD R3, R2, R1</td>
<td>W</td>
</tr>
<tr>
<td>LOAD (R2), J12(R3)</td>
<td>W</td>
</tr>
<tr>
<td>STORE (R2), 16(R2)</td>
<td>W</td>
</tr>
</tbody>
</table>

RAW data hazards
4. Control Hazards - branching causes problems since the pipeline can be filled with the wrong instructions.

IF: BEQ R3, R3, ELSE

ADD R4, R5, R6

SUB R8, R5, R6

ELSE: MUL R3, R3, R2

END_IF:

*MUL should not be executed if the previous B executes*/

END_IF:

a) During which stage is the target address (addr. of "ELSE" label) calculated for the BEQ instruction? Decode

b) During which stage of BEQ instruction is the comparison between registers (R3 and R8) performed (i.e., when is the outcome (taken or not taken) of the branch known)? E

If we always (statically) continue to fetch sequentially until the outcome of a conditional branch is known:

c) How many cycle branch penalty for a taken outcome? 2 cycles

d) How many cycle branch penalty for a not-taken outcome? 0 cycles

Branch Prediction - predict whether the branch will be taken and fetch accordingly

Static Techniques:
a) Predict never taken - continue to fetch sequentially. If the branch is not taken, then there is no wasted fetches.
b) Predict always taken - fetch from branch target as soon as possible.
(From analyzing program behavior, > 50% of branches are taken.)
c) Predict by opcode - compiler helps by having different opcodes based on likely outcome of the branch

Consider the HLL constructs:

While ($x < 0$) do

{loop body}

end while

END_WHILE:

Studies have found about a 75% successful prediction rate using this technique.

5. Suppose that you are writing a compiler for a machine that has opcodes to statically predict whether or not branches will be taken (BEQ, BEQ_PREDICT_TAKEN, BEQ_PREDICT_NOT_TAKEN, etc.). For each of the following HLL statements, predict whether or not the compiler should predict taken or not. (Briefly justify your answer)

a) integer $x$

if ($x > 0$) then

end if

b) integer $x$

if ($x = 0$) then

end if

c) integer $i$

for $i = 1$ to 500 do

end for

d) char $ch$

if ($ch >= 'a$' and $ch <= 'z$') then

end if

NOT TAKEN

TAKEN

NOT TAKEN

?
Computer Arch

Dynamic Techniques: try to improve prediction by recording program's history of conditional branch.

Problem: How do we avoid always fetching the instruction after the branch?

Instr. Addr.
400 WHILE
404 BEQ R3, R8, END WHILE
ADD R4, R5, R6

Need target of branch, but it's not calculated yet!
Plus, how do we know that we have just fetched a branch since it has not been decoded yet?

436
440 END WHILE:

Solution: Branch-prediction buffer (BPP), Branch-History Table (BHT): small, fully-associative cache to store information about most recently executed branch instructions. In a BPP, the Branch instruction address acts as the tag since that's what you know about an instruction at stage F. During the F stage, the Branch-prediction buffer is checked to see if the instruction being fetched is a branch (e.g., if the PC matches an address in the BPP).

<table>
<thead>
<tr>
<th>Valid Bit</th>
<th>Branch Instruction Address (tag field)</th>
<th>Target Address</th>
<th>Prediction of Branch</th>
<th>Prediction Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>400</td>
<td>440</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>436</td>
<td>400</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If the instruction is a branch instruction and it is in the Branch-prediction buffer, then the target address and prediction can be supplied by the BPP by the end of F for the branch instruction.

6. If the branch instruction is in the Branch-prediction buffer, will the target address supplied correspond to the correct instruction to be executed next? __No__

7. What if the instruction is a branch instruction and it is not in the Branch-prediction buffer? __Continue to fetch__

8. Should the Branch-prediction buffer contain entries for unconditional J as well as conditional branch instructions? __YES__

The table below shows the advantage of using a Branch-prediction buffer to improve accuracy of the branch prediction. It shows the impact of past n branches on prediction accuracy. Typically, two prediction bits are used so that two wrong predictions in a row are needed to change the prediction -- see above state diagram.

<table>
<thead>
<tr>
<th>n</th>
<th>Compiler</th>
<th>Business</th>
<th>Scientific</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64.1</td>
<td>64.4</td>
<td>70.4</td>
</tr>
<tr>
<td>1</td>
<td>91.9</td>
<td>95.2</td>
<td>86.6</td>
</tr>
<tr>
<td>2</td>
<td>93.3</td>
<td>96.5</td>
<td>90.8</td>
</tr>
<tr>
<td>3</td>
<td>93.7</td>
<td>96.6</td>
<td>91.0</td>
</tr>
<tr>
<td>4</td>
<td>94.5</td>
<td>96.8</td>
<td>91.8</td>
</tr>
<tr>
<td>5</td>
<td>94.7</td>
<td>97.0</td>
<td>92.9</td>
</tr>
</tbody>
</table>

Notice:
- the big jump in using the knowledge of just 1 past branch to predict the branch
- notice the big jump in going from using 1 to 2 past branches to predict the branch for scientific applications.

9. What types of data do scientific applications spend most of their time processing?

10. What would be true about the code for processing this type of data?
Computer Arch

Consider the nested loops:
for (i = 1; i <= 500; i++) {
  for (j = 1; j <= 100; j++) {
    <do something>
  }
}

Week 3 Monday

for_init_1:     li r3, 1  
for_compare_1:  bgt r3, 500, end_for_1

for_init_2:     li r4, 1  
for_compare_2:  bgt r4, 100, end_for_2

addi r4, r4, 1  
addi r3, r3, 1

j for_compare_2
j for_compare_1

Execution flow: (bold lines denote TAKEN branches) Branch Penalties without a Branch Prediction Buffer

<table>
<thead>
<tr>
<th>Branch Instruction</th>
<th>for 1 conditional (bgt r3, 500, ...)</th>
<th>for 2 conditional (bgt r4, 100, ...)</th>
<th>end for 2 uncond. (j for compare 2)</th>
<th>end for 1 uncond. (j for compare 1)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Penalties</td>
<td>2 x 1, 2</td>
<td>2 x 500, 1000</td>
<td>1 x 100 x 500 = 50,000</td>
<td>1 x 500 = 500</td>
<td>51,502</td>
</tr>
</tbody>
</table>

Execution flow: (bold lines denote TAKEN branches) Branch Penalties with 1-bit Branch Prediction Buffer

<table>
<thead>
<tr>
<th>Branch Instruction</th>
<th>for 1 conditional (bgt r3, 500, ...)</th>
<th>for 2 conditional (bgt r4, 100, ...)</th>
<th>end for 2 uncond. (j for compare 2)</th>
<th>end for 1 uncond. (j for compare 1)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Penalties</td>
<td>2 x 1, 2</td>
<td>2 x 499, 1998</td>
<td>1 x 1 = 1</td>
<td>1 x 1 = 1</td>
<td>2,002</td>
</tr>
</tbody>
</table>
11. Consider the following bubble sort algorithm that sorts an array numbers[1..n]:

```java
BubbleSort(int n, int numbers[])
{
    int bottom, test, temp;
    boolean exchanged = true;
    bottom = n - 2;
    while (exchanged) do
        exchanged = false;
        for test = 0 to bottom do
            if number[test] > number[test + 1] then
                temp = number[test];
                number[test] = number[test + 1];
                number[test + 1] = temp;
                exchanged = true;
            end if
        end for
        bottom = bottom - 1;
    end while
    end BubbleSort
}
```

<table>
<thead>
<tr>
<th>Part (a) answer</th>
<th>Part (b) answer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cond</strong></td>
<td><strong>NOT TAKEN</strong></td>
</tr>
<tr>
<td><strong>cond</strong></td>
<td><strong>NOT TAKEN</strong></td>
</tr>
<tr>
<td><strong>uncond</strong></td>
<td>?</td>
</tr>
</tbody>
</table>

a) Where in the code would unconditional branches be used and where would conditional branches be used?

b) If the compiler could predict by opcode for the conditional branches (i.e., select whether to use machine language statements like: "BRANCH_LE_PREDICT_NOT_TAKEN" or "BRANCH_LE_PREDICT_TAKEN"), then which conditional branches would be "PREDICT_NOT_TAKEN" and which would be "PREDICT_TAKEN"?

c) Assumptions:
- \( n = 100 \) and the numbers are initially in descending order before the bubble sort algorithm is called
- the five-stage RISC pipeline
- target addresses of all branches is known at the end of the D stage (so uncond. branch penalty of 1)
- the outcome of conditional branches is known at the end of the E stage (so cond. branch penalty of 2)
- ignore any data hazards

Under the above assumptions, answer the following questions:
Computer Arch  

Week 3 Monday

i) If fixed predict-never-taken is used by the hardware, then what will be the total branch penalty (#cycles wasted) for the algorithm? (Here assume NO branch-prediction-buffer)

\[\text{while cond} \quad \text{for loop cond} \quad \text{if cond} \quad \text{uncond end for} \quad \text{uncond while} \]

\[
\begin{align*}
\text{2x1} & \quad \text{2x100} & \quad 0 & \quad \frac{1}{10} \left(100 + 99 + 98 + 97 + 1\right) & \quad 100 + 1 \\
= 2 & \quad = 200 & \quad \text{10} & \quad \text{100} & \quad \text{100} \\
\end{align*}
\]

\[= 5352\]

ii) If a branch-prediction-buffer with one history bit per entry is used, then what will be the total branch penalty (#cycles wasted) for the algorithm? (Assume predict-not taken is used if there is no match in the branch-prediction-buffer) Explain your answer.

\[\text{while cond} \quad \text{for cond} \quad \text{if cond} \quad \text{uncond end for} \quad \text{uncond while} \]

\[
\begin{align*}
\text{2x1} & \quad 2 + (8+2) \times 99 & \quad 0 & \quad \frac{1}{10} \times 1 & \quad 1 \\
= 2 & \quad = 398 & \quad = 1 & \quad = 1 & \quad = 1 \\
\end{align*}
\]

\[= 402\]

iii) If a branch-prediction-buffer with two history bit per entry is used, then what will be the total branch penalty (#cycles wasted) for the algorithm? (BPB - wrong twice before prediction changed) Explain your answer.

\[
\begin{align*}
2x1 & \quad 2x100 & \quad 0 & \quad 1 \\
200 & \quad & \quad & \quad \\
\end{align*}
\]

\[= 204\]
Beyond RISC - goal of multiple instructions completed per clock cycle

*superscalar* (e.g., modern Intel x86, AMD processors) - multiple instructions in the same stage of execution in duplicate pipeline hardware

- Instruction Fetch - obtain “next” instruction(s) from memory (I cache)
- Instruction Decode - decode instr(s) and rename user-visible registers to avoid data hazards (WAW: write-after-write & WAR: write-after-read) introduced by out-of-order execution. Consider instruction sequence of:
  
  Instruction 1: MUL R3, R3, R5
  Instruction 2: ADDI R4, R3, R3    \[ RAW I_1 + I_2 on R3 \]
  Instruction 3: ADDI R5, R5, 8
  Instruction 4: SUB R7, R3, R4

  1. If these instructions were issued (selected to be executed) out-of-order and completed out-of-order, then:
     a) why would writing R3 in instruction 3 before reading R3’s value in instruction 2 cause a problem? (WAR)

     b) why would writing R3 in instruction 3 before writing R3 in instruction 1 cause a problem? (WAW)

     c) If we had more registers (say R33 - R64) and utilized them dynamically as the program executes (called “register renaming”), which registers could we rename to eliminate the WAR and WAW dependencies?

- Instruction *issue* - sent instruction to reservations unit associated with an appropriate execution unit (integer ALU, fl. pt. ALU, LOAD/STORE memory unit, etc.) to await execution
- Reservation station - *dispatch* instruction to execution unit when unit becomes free and all of the instruction’s operand values are known, i.e., all RAW data dependencies have cleared
- Instruction *retire* - writes results of potentially out-of-order instructions back to registers to ensure correct in-order completion. Also, communicates with the reservation stages when instruction completion frees resources (e.g., “virtual” registers used in register renaming)
Hammer RISC core pipeline at least 20 stages (some micro-ops require multiple execution stages)

- Other CISCeshell with inner RISC core
- Commit results of micro-ops to register set in original x86 program flow order
- up to 4 micro-ops dispatched per clock cycle
- micro-ops may be executed out of order
- Except micro-ops on superscalar pipeline
- Each x86 RISC instructions can be executed in any order on same program
- Real x86 RISC instructions from memory in order of same program
- Intel x86 Process (e.g., Pentium 4) Operation

TC knob - 
1.32 GB System Interface
Hardware Multithreading on a core:

- Find a way to "hide" true data dependency stalls, cache miss stalls, and branch stalls by finding instructions (from other process threads) that are independent of those stalling instructions.

- Multithreading - increase the utilization of resources on a core by allowing multiple processes (threads) to share the functional units of a single core
  - Processor must duplicate the state hardware for each thread - a separate register file, PC, instruction buffer, and store buffer for each thread
  - The caches, TLBs, BHT, BTB can be shared (although the miss rates may increase if they are not sized accordingly)
  - The memory can be shared through virtual memory mechanisms
  - Hardware must support efficient thread context switching

Possible options for Multithreading:

- Fine-grain - switch threads on every instruction issue
  - Round-robin thread interleaving (skipping stalled threads)
  - Processor must be able to switch threads on every clock cycle
  - Advantage - can hide throughput losses that come from both short and long stalls
  - Disadvantage - slows down the execution of an individual thread since a thread that is ready to execute without stalls is delayed by instructions from other threads

- Coarse-grain - switches threads only on costly stalls (e.g., L2 cache misses)
  - Advantages - thread switching doesn't have to be essentially free and much less likely to slow down the execution of an individual thread
  - Disadvantage - limited, due to pipeline start-up costs, in its ability to overcome throughput loss (pipeline must be flushed and refilled on thread switches)

- Simultaneous Multithreading (SMT) - A variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor (superscalar) to exploit both program ILP and thread-level parallelism (TLP)
  - Most superscalar processors have more machine level parallelism than most programs can effectively use (i.e., than have ILP)
  - With register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to dependencies among them
    - Need separate rename tables (ROBs) for each thread
    - Need the capability to commit from multiple threads (i.e., from multiple ROBs) in one cycle
  - Intel's Pentium processors are SMT, called hyperthreading, that supports just two threads (doubles the architecture state)
Threading on a 4-way SS Processor Example

<table>
<thead>
<tr>
<th>Issue slots</th>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread C</th>
<th>Thread D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
<tr>
<td></td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
</tbody>
</table>

Coarse MT

Fine MT

SMT

Intel Hyperthreading

2-way