In this homework, you will analyze a subset of the SPEC 2006 benchmark suite compiled for the ia32 (32-bit Intel architecture) ISA. Specifically, you will instrument each SPEC 2006 application binary with PIN and extract the percentages of various different instruction types. You will also use a very simple cycle accounting model to calculate the CPI for each application. Additionally, you will calculate the instruction and data footprints of each application. Finally, you will understand a few properties of the ia32 ISA.

In this assignment, we will be interested only in the following instruction types. At a very high level, we have two instruction types.

Type A: Instructions with no memory operand
Type B: Instructions with at least one memory operand

We would like to categorize instructions of type A into the following fifteen classes. The internal category names used by PIN are shown in parentheses.

1. NOPs (XED_CATEGORY_NOP)
2. Direct calls (XED_CATEGORY_CALL)
3. Indirect calls (XED_CATEGORY_CALL)
4. Returns (XED_CATEGORY_RET)
5. Unconditional branches (XED_CATEGORY_UNCOND_BR)
6. Conditional branches (XED_CATEGORY_COND_BR)
7. Logical operations (XED_CATEGORY_LOGICAL)
8. Rotate and shift (XED_CATEGORY_ROTATE | XED_CATEGORY_SHIFT)
9. Flag operations (XED_CATEGORY_FLAGOP)
10. Vector instructions (XED_CATEGORY_AVX | XED_CATEGORY_AVX2 | XED_CATEGORY_AVX2GATHER | XED_CATEGORY_AVX512)
11. Conditional moves (XED_CATEGORY_CMOV)
12. MMX and SSE instructions (XED_CATEGORY_MMX | XED_CATEGORY_SSE)
13. System calls (XED_CATEGORY_SYSCALL)
14. Floating-point (XED_CATEGORY_X87_ALU)
15. Others: whatever is left

You should only instrument the instructions that actually execute i.e., have true predicates. As a result, you should always use INS_InsertPredicatedCall method when classifying instructions.

To identify the instructions with the category shown within parentheses in the aforementioned list, you should use the INS_Category method and compare the return value against the category type e.g., to identify if an instruction ins is a NOP, you should do the following.

```
if (INS_Category(ins) == XED_CATEGORY_NOP) {
   // Increment NOP count by one
}
```

To identify direct and indirect calls, you should do the following.

```
if (INS_Category(ins) == XED_CATEGORY_CALL) {
   if (INS_IsDirectCall(ins)) {
      // Increment direct call count by one
   }
   else {
      // Increment indirect call count by one
   }
}
```

To identify instructions that have multiple sub-categories, you should do the following.

```
if ((INS_Category(ins) == XED_CATEGORY_ROTATE) || (INS_Category(ins) == XED_CATEGORY_SHIFT)) {
   // Increment rotate and shift count by one
}
```

Next, let's turn to the type B instructions. Each memory operand in such an instruction can be read and/or written to. We will view each such load or store operation within a type B instruction as a separate micro-instruction and count it as a separate instruction. For example, a type B instruction may have two load operations and a store operation. Further, each such load or store operation may access an arbitrary amount of data. However, in a 32-bit processor, it is not possible to transfer more than 32 bits of data in one shot. As a result, we will further break down each memory operation into several memory accesses of size at most 32 bits. For example, if a load operation accesses 101 bytes of data, it will be counted as 26 load operations. Finally, the instruction must be using these memory operands to carry out some operation of type A. This operation should be counted as a separate type A instruction and should also be categorized into one of the fifteen categories mentioned above.

Let us take an example of an x86 instruction that has three memory operands, two of which are loads and one is a store. The loads access one and ten bytes, respectively. The store accesses eleven bytes. This instruction will be accounted as follows.

```
Number of loads:                4
Number of stores:               3
Number of type A instruction:   1
```

Total instruction count increases by eight, whenever such an instruction is encountered.

We have discussed in the class how to identify the load and store operands of each memory instruction. To get the size of memory accessed by each operand, you should use the INS_MemoryOperandSize method.

PART A [25 points]

For each benchmark application, you need to report the dynamic counts and percentages of the following seventeen types of instructions. The total number of instructions (to be used as the denominator in the percentage) is the addition of all these seventeen counters.

1. Loads
2. Stores
3. NOPs
4. Direct calls
5. Indirect calls
6. Returns
7. Unconditional branches
8. Conditional branches
9. Logical operations
10. Rotate and shift
11. Flag operations
12. Vector instructions
13. Conditional moves
14. MMX and SSE instructions
15. System calls
16. Floating-point
17. The rest

Avoid double-counting an instruction in multiple categories. For this purpose, your instrumentation code should have the following structure.

```
if (memory instruction) {
   // Count load and store instructions for type B
}

// Categorize all instructions for type A
if (NOP) {
...
```

```
}
else if (Calls) {
   if (Direct call) {
   ...
   }
   else {
   ...
   }
}
else if (Returns) {
...
}
...
else if (Floating-point) {
...
}
else {
...
}
```

In other words, go exactly in the order shown above. Identify and categorize instructions at instrumentation stage and pass a pointer to the appropriate counter. Do not try to run a switch-case or if-elseif-else chain at analysis time. That would be so slow that you may not have enough time to finish the assignment. If you can, do the instrumentation at basic block level to speed up analysis. Prepare a table showing these counts and percentages for the applications.

PART B [10 points]

The second part of the assignment involves calculating the CPI for each benchmark. You should charge each load and store operation a fixed latency of seventy cycles and every other instruction a latency of one cycle. Tabulate the CPI of the applications in a table.

PART C [35 points]

The third part of the assignment requires you to calculate the memory footprint of each benchmark application. The memory footprint of an application is the amount of memory the application uses when executing. Note that the footprint includes both instruction and data footprints. Usually, memory footprint is calculated assuming a certain granularity G of each access. This means that any access within the G bytes will be assumed to have accessed the entire G bytes even if certain parts of the G bytes may never be touched by the application. In this assignment, we will calculate the memory footprint at a granularity of 32 bytes. For each application benchmark, you need to calculate the number of unique 32-byte instruction and data chunks accessed. Report the instruction and data statistics separately. As an example, consider an application with the following memory accesses. Each element in the sequence has two tuples separated by a colon. The first tuple specifies the instruction address and the size of the instruction in bytes. The second tuple specifies the data address and the amount of data accessed in bytes.

(1000, 4):(0, 4), (1004, 4):(4, 4), (1008, 4):(8, 4), (1012, 2):(500, 2), (1014, 6):(200, 8), (1020, 1):(220, 8), (1021, 4):(80, 8).

This application touches five different 32-byte data chunks starting at addresses 0 (the first three accesses), 64 (the last access), 192 (the fifth and part of the sixth accesses), 224 (part of the sixth access), 480 (the fourth access).
The application touches two instruction chunks starting at addresses 992 and 1024.
Thus the application has a data footprint of 160 bytes (5*32 bytes) and instruction footprint of 64 bytes (2*32 bytes). Note that in this example, we have shown instructions involving at least one memory operand. Keep in mind to include instructions of type A also in your instruction footprint. You can get the size of an instruction by calling INS_Size method. When calculating the instruction footprint, you

should not use INS_InsertPredicatedCall because even the instructions with
false predicates do get fetched, decoded, and executed up to the point of examining the predicate.
So, you should count all instructions when calculating the instruction footprint.
Prepare a table showing the instruction and data footprints of the applications.

PART D [30 points]

In this part, you will explore the following properties of the ia32 ISA.
Tabulate the results for the applications appropriately.

1. Distribution of instruction length

Report the number of instructions of various lengths (1, 2, ... bytes)
in each benchmark applications. Use the INS_Size method to get the size of each instruction. Do
not use INS_InsertPredicatedCall because you should take into account all
instructions.

2. Distribution of the number of operands in an instruction

Report the number of instructions with 0, 1, 2, ... operands. Use the INS_OperandCount
method to get the number of operands in an instruction. Do not use INS_InsertPredicatedCall
because you should take into account all instructions.

3. Distribution of the number of register read operands in an instruction

An instruction reads a certain number of source registers to get the operand values.
Report the number of instructions with 0, 1, 2, ... register read operands.
Use the INS_MaxNumRRegs method to get the number of register read operands of
an instruction. Do not use INS_InsertPredicatedCall because you should take
into account all instructions.

4. Distribution of the number of register write operands in an instruction

An instruction writes to a certain number of destination registers.
Report the number of instructions with 0, 1, 2, ... register write operands.
Use the INS_MaxNumWRegs method to get the number of register write operands of
an instruction. Do not use INS_InsertPredicatedCall because you should take
into account all instructions.

5. Distribution of the number of memory operands in an instruction

Report the number of instructions with 0, 1, 2, ... memory operands.
Instrument only the instructions with true predicates i.e., use
INS_InsertPredicatedCall. Note that a memory operand that is read as well as
written to within the same instruction, should be counted as two different
operands.

6. Distribution of the number of memory read operands in an instruction

Report the number of instructions with 0, 1, 2, ... memory read operands.
Instrument only the instructions with true predicates i.e., use INS_InsertPredicatedCall.

7. Distribution of the number of memory write operands in an instruction

Report the number of instructions with 0, 1, 2, ... memory write operands.
Instrument only the instructions with true predicates i.e., use INS_InsertPredicatedCall.

8. Report the maximum and average number of memory bytes touched (read and written)
by any memory instruction. The denominator in the average should include only those
instructions that have at least one memory operand. Instrument only the instructions
with true predicates i.e., use INS_InsertPredicatedCall.

9. Report the maximum and minimum values of the immediate field in an instruction.
Recall that in an instruction each operand may or may not use the immediate mode.
To test if an operand is in immediate mode, use the INS_OperandIsImmediate method.

The immediate value for an operand is relevant only if INS_OperandIsImmediate returns true. Use the INS_OperandImmediate method to get the immediate value in such a case. The return type of this method for a 32-bit binary is a signed 32-bit integer i.e., INT32. Since there is no IARG_INT32 type in PIN, you can pass the immediate value from the instrumentation routine to the analysis routine through IARG_ADDRINT. Make sure to use INT32 type in the analysis routine to recover the sign. Do not use INS_InsertPredicatedCall because you should take into account all instructions.

10. Report the maximum and minimum values of the displacement field in a memory instruction. Use the INS_OperandMemoryDisplacement method to get the value of the displacement field in a memory operand. Every ia32 memory operand can use a displacement+base+index*scale addressing. The memory displacement is of type ADDRDELTA which you can pass from from the instrumentation routine to the analysis routine through IARG_ADDRINT. Make sure to use ADDRDELTA type in the analysis routine to recover the sign. Instrument only the instructions with true predicates i.e., use INS_InsertPredicatedCall.

Please consult the following page to learn about the PIN API and the syntax of the methods you will be using.

https://software.intel.com/sites/landingpage/pintool/docs/98749/Pin/doc/html/group__API__REF.html

---
SETUP

Benchmark applications

You will be using the following eight applications drawn from the SPEC 2006 benchmark suite for this homework:

1. 400.perlbench diffmail.pl
2. 401.bzip2 input.source
3. 403.gcc cp-decl.i
4. 429.mcf
5. 450.soplex ref.mps
6. 456.hmmer nph3.hmm
7. 471.omnetpp
8. 483.xalancbmk

Optional (caution: executions take more than twelve hours):

9. 436.cactusADM
10. 437.leslie3d
11. 462.libquantum
12. 470.lbm
13. 482.sphinx3

The input and binary for each of the above applications are provided in a separate compressed archive spec_2006.zip. Download the archive from:
https://www.cse.iitk.ac.in/users/mainakc/2023Autumn/spec_2006.zip
Use the following command to unzip the archive:

```
unzip spec_2006.zip
```

This would create a directory spec_2006. The directory spec_2006 contains a file commandline.txt and a sub-directory for each application. The sub-directory of each application contains its binary and inputs. The file commandline.txt provides the commandline to run the applications natively.

For each benchmark application, you will fast-forward it over a certain number of instructions (specified in the file commandline.txt). The fast-forwarding amount is decided using the SimPoint tool.
During this phase do not invoke any analysis code. After this phase, analyze the application for a total
of one billion (10^9) instructions (includes all instructions in this count, even those with false predicate)

and then report the statistics.

Fast-forwarding can be implemented by keeping track of the number of instructions executed (even those
with false predicates), and then checking in analysis routines when sufficient instructions have been executed. We have already discussed in class how to track the number of instructions executed.

Below I show an extension of the instruction count PIN tool discussed in class for fast-forwarding.

```
UINT64 fast-forward-count;      // Should be a command line input to your PIN tool
UINT64 icount = 0;

// Analysis routine to track number of instructions executed
void InsCount() { icount++; }

void MyPredicatedAnalysis(...) {
        // if fast-forward number of instructions have been executed and one billion
        // instructions after fast-forward have not been executed, then do the analysis
        if ((icount >= fast-forward-count) && (icount < fast-forward-count + 1,000,000,000))
        {
                // analysis code
        }
}

// Instrumentation routine
Ins_InsertCall(ins, IPOINT_BEFORE, InsCount, IARG_END);
Ins_InsertPredicatedCall(ins, IPOINT_BEFORE, MyPredicatedAnalysis, ..., IARG_END);
```

The above implementation of fast-forwarding is not that efficient, as PIN cannot inline the the MyPredicatedAnalysis routine due to the conditional control flow in it.

This is a very common case, where an analysis routine has a single "if-then" test, and a small amount of analysis code plus the test is always executed but the "then" part is executed only once in a while.

To inline this common case, PIN provides a set of conditional instrumentation APIs for the tool writer to rewrite their analysis routines into a form that does not have control-flow changes. Below I show how to rewrite the same fast-forwarding logic using the conditional instrumentation APIs of PIN.

```
// Analysis routine to track number of instructions executed
void InsCount() { icount++; }

// Analysis routine to check fast-forward condition
ADDRINT FastForward (void) {
        return ((icount >= fast-forward-count) && (icount < fast-forward-count + 1,000,000,000));
}

// Predicated analysis routine
void MyPredicatedAnalysis(...) {
        // analysis code
}

// Instrumentation routine
Ins_InsertCall(ins, IPOINT_BEFORE, InsCount, IARG_END);

// FastForward() is called for every instruction executed
INS_InsertIfCall(ins, IPOINT_BEFORE, FastForward, IARG_END);

// MyPredicatedAnalysis() is called only when the last FastForward() returns a non-zero value.
INS_InsertThenPredicatedCall(ins, IPOINT_BEFORE, MyPredicatedAnalysis, ..., IARG_END);
```

As SPEC benchmark applications are long running applications we would like to stop the application, once we have measured statistics for 1 billion instructions after the fast-forwarding. Below I show how to do this with fast forwarding using the

```
conditional instrumentation APIs of PIN.

// Analysis routine to track number of instructions executed, and check the exit condition
ADDRINT InsCount()      {
        icount++;
}

ADDRINT Terminate(void)
{
        return (icount >= fast-forward-count + 1,000,000,000);
}

// Analysis routine to check fast-forward condition
ADDRINT FastForward(void) {
        return (icount >= fast-forward-count && icount);
}

// Analysis routine to exit the application
void MyExitRoutine(...) {
        // Do an exit system call to exit the application.
        // As we are calling the exit system call PIN would not be able to instrument application
end.
        // Because of this, even if you are instrumenting the application end, the Fini function
would not
        // be called. Thus you should report the statistics here, before doing the exit system call.

        // Results etc
        exit(0);
}

// Predicated analysis routine
void MyPredicatedAnalysis(...) {
        // analysis code
}

// Instrumentation routine
INS_InsertIfCall(ins, IPOINT_BEFORE, (AFUNPTR) Terminate, IARG_END);

// MyExitRoutine() is called only when the last call returns a non-zero value.
INS_InsertThenCall(ins, IPOINT_BEFORE, MyExitRoutine, ..., IARG_END);

// FastForward() is called for every instruction executed
INS_InsertIfCall(ins, IPOINT_BEFORE, FastForward, IARG_END);

// MyPredicatedAnalysis() is called only when the last FastForward() returns a non-zero value.
INS_InsertThenPredicatedCall(ins, IPOINT_BEFORE, MyPredicatedAnalysis, ..., IARG_END);

// Instrumentation routine
Ins_InsertCall(ins, IPOINT_BEFORE, InsCount, IARG_END);

Note on basic block-level instrumentation:

If you want to do instrumentation at the level of basic blocks, some approximations may
be necessary. Since the fast-forward may not end at a basic block boundary, you may overshoot
the fast-forward amount a little bit. Similarly, the execution of one billion instructions may
not end at a basic block boundary. So, you may overshoot this too. I will accept such an
implementation also.

Setting up PIN environment and PIN tool
---------------------------------------
Download the latest revision of pinkit (Pin 3.28) for the Linux operating
system from the PIN home. Extract the pinkit using the following command:
        tar -xvzf pin-3.28-98749-g6643ecee5-gcc-linux.tar.gz.tar.gz
This would create the pin-3.28-98749-g6643ecee5-gcc-linux directory, which contains the pinkit.
```

```
Pinkit provides a sample setup for building custom tools in the source/tools/MyPinTool directory.
The main benefit of using this setup is that it already contains makefiles for compiling PIN tools.
You should use this sample setup for building your PIN tools.

Below I describe how to build a PIN tool called "HW1" using the sample setup:
1. Create a HW1 directory in the source/tools/ directory of the pinkit:
        cd pin-3.28-98749-g6643ecee5-gcc-linux/source/tools/
        mkdir HW1

2. Copy MyPinTool.cpp, makefile and makefile.rules files from the source/tools/MyPinTool directory
to HW1 directory:
        cd HW1/
        cp ../MyPinTool/MyPinTool.cpp ./
        cp ../MyPinTool/makefile ./
        cp ../MyPinTool/makefile.rules ./

3. Rename the MyPinTool.cpp file in HW1 directory to HW1.cpp:
        mv MyPinTool.cpp HW1.cpp

4. Edit line no 20 of makefile.rules file in HW1 directory from "TEST_TOOL_ROOTS := MyPinTool"
to "TEST_TOOL_ROOTS := HW1"

5. The file HW1.cpp is the source file of the HW1 PIN tool. Write your PIN tool in this file.

6. Once you have written the PIN tool, you can build it using the make command as follows:
        #On a 32-bit system use the following make command:
        make obj-ia32/HW1.so

        #On a 64-bit system use to following make command:
        make TARGET=ia32 obj-ia32/HW1.so

This would create the obj-ia32 directory which contains the binary HW1.so for HW1 PIN tool.
As we will be instrumenting 32-bit SPEC benchmark binaries we should create a 32-bit version
of PIN tool. Thus on a 64-bit system we need to provide the extra TARGET=ia32 argument to make.
To find out whether your Linux machine is a 32-bit or a 64-bit system run the following command
on terminal:
        uname -m
If the output is "x86_64" then the machine is a 64-bit machine and if the output is "ia32" or
"i686" then the machine is a 32-bit system.

To enable debugging support (only if needed), supply DEBUG=1 option to make while building PIN tools,
as shown below:
        #On a 32-bit system use the following make command:
        make DEBUG=1 obj-ia32/HW1.so

        #On a 64-bit system use to following make command:
        make TARGET=ia32 DEBUG=1 obj-ia32/HW1.so

7. Notice how knobs are used in HW1.cpp to pass command line arguments to the PIN tool. Your PIN
tool will have two command line arguments, namely, the fast-forward amount and the output
file name where all results should be written to at the end.

8. To run the PIN tool with the SPEC binaries, you first need to change directory to the
application which you want to analyze and then invoke PIN. For example,
        cd ~/spec_2006/400.perlbench/
        ~/pin-3.28-98749-g6643ecee5-gcc-linux/pin -t ~/pin-3.28-98749-g6643ecee5-gcc-
linux/source/tools/HW1/obj-ia32/HW1.so -f 207 -o perlbench.diffmail.out -- ./perlbench_base.i386 -
I./lib diffmail.pl 4 800 10 17 19 300 > perlbench.ref.diffmail.out 2> perlbench.ref.diffmail.err

Notice that I pass the fast-forward amount in billions of instructions using -f 207.

If you want to test your tool for smaller binaries such as /bin/ls, make sure to compile your PIN
tool
for the native ISA i.e., if you are running on a 64-bit machine, you should compile the PIN
tool for 64 bits as we did in the class. This is because /bin/ls would have been compiled for 64 bits
```

```
on such a machine.
        ~/pin-3.28-98749-g6643ecee5-gcc-linux/pin -t ~/pin-3.28-98749-g6643ecee5-gcc-
linux/source/tools/HW1/obj-intel64/HW1.so -o ls-stats.out -- /bin/ls

9. Your PIN tool will require at least the following analysis calls: one for keeping count of
instructions executed, one for exit condition check, one for exiting the application,
one for fast-forwarding condition check, one for instrumenting instructions with
only true predicates, and one for instrumenting all instructions:

INS_InsertIfCall(ins, IPOINT_BEFORE, Terminate, IARG_END);
INS_InsertThenCall(ins, IPOINT_BEFORE, MyExitRoutine, ..., IARG_END);

INS_InsertIfCall(ins, IPOINT_BEFORE, FastForward, IARG_END);
INS_InsertThenPredicatedCall(ins, IPOINT_BEFORE, MyPredicatedAnalysis, ..., IARG_END);

INS_InsertIfCall(ins, IPOINT_BEFORE, FastForward, IARG_END);
INS_InsertThenCall(ins, IPOINT_BEFORE, MyAnalysis, ..., IARG_END);

INS_InsertCall(ins, IPOINT_BEFORE, InsCount, IARG_END);

WHAT TO SUBMIT
--------------

Prepare a PDF document containing the result tables. Share any additional information
about your implementation that you would like to share. Put the document in your HW1
directory. Please note that we will not accept anything other than a PDF file.
Archive your HW1 directory after removing all .o, .so files, and obj-* directories:

zip -r HW1_MYROLLNO.zip HW1/

Please note that we will not accept anything other than a .zip file. Replace
MYROLLNO by your roll number. Send HW1_MYROLLNO.zip to
cs422autumn2023@gmail.com with subject line [CS422 HW1].
```