1. Purpose

You are to build a superscalar implementation of a 16-bit microarchitecture. The purpose of this assignment is to get you to think about the problems of designing high-performance processors. It will not be graded; however, there will be two prizes for the following outstanding entries:

1. **The simulator that performs the best in terms of cycles per instruction**
   This means the simulator that executes the test programs in the fewest number of simulated cycles, which corresponds to the fastest processor, if we were building chips.

2. **The simulator that performs the best in terms of cycles per real-second**
   This means the simulator that executes the test programs in the fewest number of seconds, as measured in real-time, which corresponds to the fastest simulator, i.e. the simulator that is designed to be the most efficient in its execution.

You have two weeks: simulators are due **Monday, February 11**. Collaboration is encouraged. The prizes will be a trophy for each winning team and free pizza, as well as bragging rights for the rest of the semester.

2. The RiSC-16: A Ridiculously Simple Computer

You will use the instruction set of the RiSC-16, an imaginary instruction-set architecture that was introduced to the ENEE 350 class last semester. The RiSC-16 is an 8-register, 16-bit computer. All addresses are shortword-addresses (i.e. address 0 corresponds to the first two bytes of main memory, address 1 corresponds to the second two bytes of main memory, etc.). Register 0 will always contain the value 0; this should be enforced by hardware. The RiSC-16 is very simple, but it is general enough to solve complex problems.

There are 4 machine-code instruction formats: RRR-type, RRI-type, RR-type, and RI-type.

<table>
<thead>
<tr>
<th>Bit:</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRR-type:</td>
<td>3 bits</td>
<td>3 bits</td>
<td>3 bits</td>
<td>4 bits</td>
<td>3 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opcode</td>
<td>reg A</td>
<td>reg B</td>
<td>0</td>
<td>reg C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RRI-type:</td>
<td>3 bits</td>
<td>3 bits</td>
<td>3 bits</td>
<td>7 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opcode</td>
<td>reg A</td>
<td>reg B</td>
<td>signed immediate (-64 to 63)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR-type:</td>
<td>3 bits</td>
<td>3 bits</td>
<td>3 bits</td>
<td>7 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opcode</td>
<td>reg A</td>
<td>reg B</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RI-type:</td>
<td>3 bits</td>
<td>3 bits</td>
<td>10 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opcode</td>
<td>reg A</td>
<td>unsigned immediate (0 to 1023)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The following table describes the different instruction operations.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name and Format</th>
<th>Opcode (binary)</th>
<th>Assembly Format</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Add RRR-type</td>
<td>000</td>
<td>add rA, rB, rC</td>
<td>Add contents of regB with regC, store result in regA.</td>
</tr>
<tr>
<td>addi</td>
<td>Add Immediate RRI-type</td>
<td>001</td>
<td>addi rA, rB, imm</td>
<td>Add contents of regB with imm, store result in regA.</td>
</tr>
<tr>
<td>nand</td>
<td>Nand RRR-type</td>
<td>010</td>
<td>nand rA, rB, rC</td>
<td>Nand contents of regB with regC, store results in regA.</td>
</tr>
<tr>
<td>lui</td>
<td>Load Upper Immediate RI-type</td>
<td>011</td>
<td>lui rA, imm</td>
<td>Place the 10 ten bits of the 16-bit imm into the 10 ten bits of regA, setting the bottom 6 bits of regA to zero.</td>
</tr>
<tr>
<td>lw</td>
<td>Load Word RRI-type</td>
<td>100</td>
<td>lw rA, rB, imm</td>
<td>Load value from memory into regA. Memory address is formed by adding imm with contents of regB.</td>
</tr>
<tr>
<td>sw</td>
<td>Store Word RRI-type</td>
<td>101</td>
<td>sw rA, rB, imm</td>
<td>Store value from regA into memory. Memory address is formed by adding imm with contents of regB.</td>
</tr>
<tr>
<td>beq</td>
<td>Branch If Equal RRI-type</td>
<td>110</td>
<td>beq rA, rB, imm</td>
<td>If the contents of regA and regB are the same, branch to the address PC+1+imm, where PC is the address of the beq instruction.</td>
</tr>
<tr>
<td>jalr</td>
<td>Jump And Link Register RR-type</td>
<td>111</td>
<td>jalr rA, rB</td>
<td>Branch to the address in regB. Store PC+1 into regA, where PC is the address of the jalr instruction.</td>
</tr>
</tbody>
</table>

3. **RiSC-16 Assembly Language and Assembler**

I will provide you with an assembler for the RiSC-16. It is called “a” and can be found in the class directory:

/afs/glue.umd.edu/department/enee/public_html/courses/enee759m.S2002/bin

You will probably want to put this path in your $path variable. Note that this assembler will only run on SPARC-based machines. I will provide you with the source code should you wish to recompile the assembler for some other architecture (e.g. x86).

The format for a line of assembly code is:

```
label:<whitespace>opcode:<whitespace>field0, field1, field2:<whitespace>#: comments
```

The leftmost field on a line is the label field. Valid RiSC labels are any combination of letters and numbers followed by a colon. The colon at the end of the label is not optional—a label without a colon is interpreted as an opcode. After the optional label is whitespace (space/s or tab/s). Then follows the opcode field, where the opcode can be any of the assembly-language instruction mnemonics listed in the above table. After more whitespace comes a series of fields separated by commas and possibly whitespace (the whitespace is not necessary; the commas are). All fields are
given as decimal numbers. The number of fields depends on the instruction. The following table describes the instructions.

<table>
<thead>
<tr>
<th>Assembly-Code Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>lui regA, immed</td>
<td>R[regA] &lt;- immed &amp; 0xffc0</td>
</tr>
</tbody>
</table>

Anything after a pound sign (‘#’) is considered a comment and is ignored. The comment field ends at the end of the line. Comments are vital to creating understandable assembly-language programs, because the instructions themselves are rather cryptic.

In addition to RiSC instructions, an assembly-language program may contain directives for the assembler. These are often called pseudo-instructions. The six assembler directives we will use are nop, halt, lli, movi, .fill, and .space (note the leading periods for .fill and .space).

<table>
<thead>
<tr>
<th>Assembly-Code Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>do nothing</td>
</tr>
<tr>
<td>halt</td>
<td>stop machine &amp; print state</td>
</tr>
<tr>
<td>movi regA, immed</td>
<td>R[regA] &lt;- immed</td>
</tr>
<tr>
<td>.fill immed</td>
<td>initialized data with value immed</td>
</tr>
<tr>
<td>.space immed</td>
<td>data array of size immed</td>
</tr>
</tbody>
</table>

The following paragraphs describe these pseudo-instructions in more detail:

- The **nop** pseudo-instruction means “do not do anything this cycle” and is replaced by the instruction **add 0,0,0** (which clearly does nothing).
- The **halt** pseudo-instruction means “stop executing instructions and print current machine state” and is replaced by **beq 0,0,-1** (which clearly halts the machine—by looping forever without modifying any machine state—but is interpreted by the machine to be a special instance of **beq**).
- The **lli** pseudo-instruction (*load-lower-immediate*) means “OR the bottom six bits of this number into the indicated register” and is replaced by **addi X,X,imm6**, where **X** is the reg-
ister specified, and \texttt{imm6} is equal to \texttt{imm} \& \texttt{0x3f}. This instruction can be used in conjunction with \texttt{lui}: the \texttt{lui} first moves the top ten bits of a given number (or address, if a label is specified) into the register, setting the bottom six bits to zero, and the \texttt{lli/addi} moves the bottom six bits in. The \texttt{addi} uses a six-bit number, which is guaranteed to be interpreted as positive and thus avoids sign-extension, and the resulting add is essentially a concatenation of the two bitfields.

- The \texttt{movi} pseudo-instruction is just shorthand for the \texttt{lui+lli} combination. Note, however, that the \texttt{movi} instruction \textit{looks} like it only represents a single instruction, whereas in fact it represents two. This can throw off your counting if you are expecting a certain distance between instructions. Thus, it is always a good idea to use labels wherever possible.

- The \texttt{.fill} directive tells the assembler to put a number into the place where the instruction would normally be stored. The \texttt{.fill} directive uses one field, which can be either a numeric value or a symbolic address. For example, \texttt{.fill 32} puts the value 32 where the instruction would normally be stored. Using \texttt{.fill} with a symbolic address will store the address of the label. In the example below, the line \texttt{.fill start} will store the value 2, because the label \texttt{start} refers to address 2.

- The \texttt{.space} directive takes one integer \texttt{n} as an argument and is replaced by \texttt{n} copies of \texttt{“.fill 0”} in the code; i.e., it results in the creation of \texttt{n} 16-bit words all initialized to zero.

The following is an assembly-language program that counts down from 5, stopping when it hits 0.

```
lw 1,0,count # load reg1 with 5 (uses symbolic address)
lw 2,1,3   # load reg2 with -1 (uses numeric address)
start: add 1,1,2   # decrement reg1 -- could have been addi 1,1,-1
        beq 0,1,2   # goto end of program when reg1==0
        beq 0,0,start # go back to the beginning of the loop
        nop
done:  halt # end of program
count: .fill 5
neg1:  .fill -1
startAddr: .fill start # will contain the address of start (2)
```

The main goal of this example is to illustrate the difference between using labels and integers in the immediate fields, as all other instructions should be relatively self-explanatory. Here is the corresponding machine-level interpretation of the program:

```
(address 0): 8407 (signed decimal -31737, unsigned decimal 33799)
(address 1): 8883 (signed decimal -30589, unsigned decimal 34947)
(address 2): 0482 (signed decimal  1154, unsigned decimal 1154)
(address 3): c082 (signed decimal -16254, unsigned decimal 49282)
(address 4): c07d (signed decimal -16259, unsigned decimal 49277)
(address 5): 0000 (signed decimal  0, unsigned decimal 0)
(address 6): c07f (signed decimal -16257, unsigned decimal 49279)
(address 7): 0005 (signed decimal  5, unsigned decimal 5)
(address 8): ffff (signed decimal  -1, unsigned decimal 65535)
(address 9): 0002 (signed decimal  2, unsigned decimal 2)
```

The following is the program’s representation in an executable file:

```
8407
8883
0482
c082
c07d
0000
c07f
0005
fff
0002
```

Note that the format for executable RiSC-16 programs looks just like this: ASCII hexadecimal numbers, one per line. This allows the RiSC-16 executables to be completely portable across dif-
different machine types: i.e., a C-code simulator for the RiSC-16 should be easily portable between little-endian and big-endian machines.

In general, acceptable RiSC assembly code is one-instruction-per-line. It is okay to have a line that is blank, whether it is commented out (i.e., the line begins with a pound sign) or not (i.e., just a blank line). However, a label cannot appear on a line by itself; it must be followed by a valid instruction on the same line (a .fill directive or halt/nop/etc counts as an instruction).

4. Example simulator

I will provide you with the C source code for an example simulator. It is fully pipelined, and it employs direct-mapped split level-1 caches, dynamic branch prediction, and 2-way out-of-order superscalar issue. You are more than welcome to use this source code as a starting point, or you may start from scratch. Either way, your implementation must reflect a pipelined organization.

4.1 Caches & Memory System

The example uses a Harvard architecture cache system (split instruction and data caches). The caches are write-through, write-allocate. The cache size for each is 64 bytes, the blocksize is two words (4 bytes), and the cache’s associativity is 2. There is a unified main memory system immediately beneath the caches; when the program is read into the simulator, it is read into main memory but not the caches—the caches must be initialized to zero. When data is found in the cache, there is no penalty. When data is not found in the cache, we fetch the data from main memory; this incurs a significant penalty because the caches are blocking. The entire pipeline stalls (including stages after the memory stage) until the data arrives from main memory 100 cycles later—this is easily done by adding 100 to the current cycle counter. We simplify the design of the caches enormously by assuming that the bus between the processor and main memory is as wide as the block size; therefore the entire block is fetched in the same cycle, which means that other words within the block are available immediately. In real systems, the next word in the block may or may not be available on the very next cycle, so if two back-to-back loads reference successive words from main memory, the second load might very well have to stall several cycles until the rest of the cache line arrives. When you see timing numbers like the following for commercial systems:

12-1-1-1

this indicates that the first part of the cache block arrives in 12 bus cycles, the next part arrives 1 cycle later, the next arrives one cycle after that, etc.

4.2 Dynamic Branch Prediction

This example uses a 2-level branch prediction scheme and a branch-target buffer to predict both the branch’s direction and its target. We use a 4-bit pattern history to index a set of 16 2-bit saturating counters. The value in the pattern history register is updated when we know the outcome of the branch (occurs late in the pipeline). If the branch was taken, we shift the history bits left by one and add a 1 to the end. If the branch was not-taken, we shift the history bits left by one and add a 0 to the end. Therefore if a branch with a given PC was taken once, then not-taken three times, the value in the history register should be 1000. If the branch is then taken on the next pass, the value in the history register should become 0001.

The problem with branch prediction is that, unless you lengthen the cycle time so that you can access the instruction cache, perform an addition, and complete a 4-way mux all in one cycle, you do not know the predicted branch target until the ID stage, which means if you predict branch
taken, you have to insert a stall into the pipeline. Thus, you will often hear of a “1-cycle branch taken penalty” in many processors. A solution is to predict not only the direction of the branch (which can be done in parallel with the instruction-cache access), but to predict the target of the branch as well—which can also be done in parallel with the instruction-cache access. Thus, at the end of the IF stage, we have predicted both the direction and the target and if the instruction turns out to be a branch (we predict these things before we know the instruction), we use the predicted values. So, as it turns out, we can obtain the branch target by the end of the IF stage.

The structure used for this is the branch target buffer, a cache of branch targets. The example implementation is direct-mapped, indexed by the PC of the branch instruction (which, of course, is available at the beginning of the IF stage). Once the actual target is known (during the ID stage), we update the branch target buffer accordingly, and set the entry’s valid bit to 1.

4.3 Dynamic Scheduling

We implement a very simple 2-issue out-of-order superscalar machine. Its issue mechanism is actually not far from the PowerPC implementation. In the IF stage, we fetch up to 4 instructions from the instruction cache into an issue buffer. On every cycle, we scan the issue buffer for up to two instructions that can be issued in parallel, which is more complicated than simply observing that there are no dependencies between the two. For instance, if instructions 1 and 3 in the buffer have no inter-dependencies, we must also verify that instruction 3 does not write to locations that instruction 2 reads. This would be an example of a WAR hazard—write-after-read—instruction 3 writes to a register after instruction 2 reads from it, so we cannot allow instruction 3 to issue ahead of instruction 2 because instruction 2 would then get the wrong data.

If we only find one instruction to issue, then we only issue one instruction, issuing a NOP instruction to the second pipeline (a NOP is the value 0, which translates to add 0,0,0). Both pipelines are identical and the data cache can handle two memory operations in a single cycle. This is not as straightforward as it sounds. The cache must handle odd cases such as when two memory operations reference the same location. If one operation stores to location X from register 1 and another operation stores to location X from register 2, it is possible that the dependency-detection logic fails to recognize this, and the two instructions are issued simultaneously. Which memory access should execute last?

We dodge the issue of interrupts and allow out-of-order instruction commit. Reorder buffers are fairly complex and we will get to them in several weeks. For the moment, you may pretend that interrupts will never happen.

4.4 Performance Measurements

We keep statistics on the following event types:

- instruction and data cache stall cycles (cycles spent waiting for main memory)
- instruction and data cache miss rates
- total cycles required to execute program
- total instructions executed
- number of branch mispredicts & stall cycles
- number of branches executed
- frequency of 2-, 1-, and 0-instruction pipeline issues
- total CPI
These are summarized at the end of the simulator’s execution.

5. **RiSC-16 Simulator Design**

You are to build the fastest machine that you can. Your L1 cache access time can be 1-cycle (pipelined), any L2 cache access time must be 10 cycles, and your memory access time must be 100 cycles. You are not allowed to perform “Sun-style execution” (i.e. cheat). Beyond this, you are allowed to scale parameters however you wish—provided that the architecture be realizable: it must be buildable. You are allowed to scale one facet of the design to unbuildable dimensions if you choose: for example, perfect caches or perfect branch prediction or 100-way superscalar or something like that.

There are no other rules. You may use whatever branch prediction you wish. You may use data prediction. You can implement as large an instruction window as you want, your pipeline organization can be anything, your decode/dispatch/issue logic can be anything. Your design can be in-order, out-of-order, whatever. You can use any organization of cache and/or prefetch buffers. Your caches can be multi-ported, as can your memory system. Your caches can be lockup-free, provided you implement appropriate data structures to handle that. In order to win the prize, you must keep track of your system’s behavior so that you can justify your performance numbers; i.e. claiming a CPI value of 0.5 without any justification will not get you any trophies or pizza. The winning design should be able to show exactly why it is the winning design.