Assignment 3

A RRISCy Startup

Assignment Due: 11:59 pm, Wednesday, September 27, 2006.
Self-Grading Due: 5:00 pm, Friday, September 29, 2006.

In this assignment, you will learn about working with pointers and primitive arrays, as well as understanding constructors, destructors, and the assignment operator. As a side benefit, you will also learn about working with a slightly more realistic codebase and perhaps gain some insights into both simulation and the operation of microprocessors beyond what you have seen in CS 60.

Scenario

Morgan Kim and Terry Harper were CS 70 students at the beginning of last semester, but quit the college to form a startup company. Observing that AMD and Intel are both selling mainstream processors with 64-bit extensions, Morgan and Terry have decided that the moment is ripe to launch a new purely 64-bit architecture, and they have left to found a startup company doing processor design.

You may have heard of Risc (Reduced Instruction Set Computer) machines; having a small number of simple instructions can allow a cleaner and faster processor design (as opposed, say to VAX computers that supported evaluate-polynomial in hardware). Morgan and Terry have decided that Risc should be pushed even further, and have started design of the Noronic™ processor, following what they call a Rrisc (Really Reduced Instruction Set Computer) philosophy. As a marketing decision, the Noronic™ is an "all 64-bit" processor that not only uses 64-bit registers for holding 64-bit data, but (unusually) also has 64-bit instructions as well. These extra-wide instructions allow the processor to support hundreds of high-speed registers, as opposed to only 32 on most Risc systems (and eight on the x86).

Morgan and Terry have come up with a preliminary set of instructions that the Noronic™ will support. To help with the remaining design decisions they have decided to write an Rrisc simulator. Although they learned some things about C++ and coding in what little they saw of CS 70, they left just before we began discussed pointer arithmetic and array `new`, and `delete`.[1] Realizing their mistake, they have turned to you. In return for several shares of stock, you and your partner have agreed to complete their implementation and run a few tests.

1. They also seem to have taken the recommendation against gratuitous use of `goto` in C++ a bit too seriously; unlike most processors, their Rrisc architecture has no specialized jump instructions.

Setup

Set up your assignment directory by executing `svn update` from your `cs70` directory. This command should check out the following files:

`hw3/Makefile`	`hw3/iounit.hpp`
`hw3/rrisc.cpp`	`hw3/iounit.cpp`
`hw3/common.hpp`	`hw3/idecode.hpp`
`hw3/cpu.hpp`	`hw3/idecode.cpp`
`hw3/cpu.cpp`	`hw3/instr.hpp`
`hw3/storage.hpp`	`hw3/instr.cpp`
`hw3/storage.cpp`	`hw3/asm.pl`
`hw3/cache.hpp`	`hw3/fact.assembly`
`hw3/cache.cpp`	`hw3/hello.assembly`
	`hw3/matrix1.assembly`
	`hw3/matrix2.assembly`

Note that this assignment concerns itself with the files listed in the left column. You will not need to change (or even understand the implementation of) the the files in the right column, which define the *IOUnit*, *InstructionDecoder*, and *Instruction* classes, as well as providing assembly-language source code and an assembler for the Noronic™.

The Noronic™ Simulator

Although it is not yet finished, Terry and Morgan's simulator for the Noronic™ processor is already functional enough to run actual programs (written in Noronic™ machine code). When you run `make`, you will build the simulator (called `rrisc`), and some machine-code programs for it to run (e.g., `hello.hex` and `fact.hex`). For example, you can run the canonical "Hello World" program by typing:

```
./rrisc hello.hex
```

You can run `fact.hex`, which calculates the factorial of `5` the same way. (Note that when the simulator terminates, it dumps the contents of the simulated machine's memory and registers, so you may need to scroll back up in your terminal window to see the point where it prints the "`Hello World`" or the calculated factorial.)

The simulator has a number of option switches to control various aspects of the simulated hardware.

`-t` *level* — Specify tracing level

There are several tracing levels that can provide an increasing level of detail. The possible values are

`0` — No tracing & no output of final CPU/memory state, execution stats, etc.

`1` — No tracing

2 — Trace the CPU core
3 — Trace access to memory
4 — Also display sub-steps of each instruction execution
5 — Dump memory and registers after every instruction

`-r` *regs* — Number of registers in the CPU

Note that the machine-code programs you have been given assume that there are 16 registers. This code will break if you run with a different number of registers, not least because the Noronic™ uses the highest numbered register to hold the program counter (i.e., where in memory to find the next instruction). Thus when you have 16 registers, the program counter is stored in register `r15` (or register location `0x0F`).

`-m` *mem* — Number of words of main memory

`-c` *cycles* — Number of CPU cycles needed to read an arbitrary word from memory

`-s` *cycles* — Number of CPU cycles needed to read a successive word from memory

`-w` *words* — The number of memory words per cache line

`-l` *lines* — The number of lines stored in the cache

Feel free to experiment with running the simulator with some different settings. For example, as you vary the memory latencies, you see different cycle counts; if you set memory too small, you won't even be able to load the program; and if you change the number of registers, the programs we have provided will fail to work properly.

Coding Component

Unfortunately, for some of the settings you will get inaccurate cycle counts, and for others the simulator will stop prematurely, because the *Memory* object always assumes a 48-word memory size (which is too small to run `matrix1.hex` and `matrix2.hex`), and the *Cache* object is only partially implemented. Your job will be to fix these deficiencies.

Coding Questions

C1. The *Storage* class is essentially just a mapping from an integer index to a machine word. Objects of this class are used both to represent a *Processor*'s hardware registers and to represent main memory. Although the first constructor is helpfully passed the number size of elements the memory should hold, the declaration of `contents_` is wrong. Morgan and Terry didn't know how to specify an array that could vary in size, and so just declared a fixed size array (storing 48 *machine_word_t*s) instead.

By putting the array of machine words onto the heap, we can choose how big it should be at run-time. Modify the declaration of `contents_` and *Storage*'s constructor(s) and destructor so that the simulated memory is a dynamically allocated array.

Modify any other code in the class necessary to make the class behave essentially as before. For example, the copy constructor should continue to create a copy of memory (rather than sharing the same array) and the simulated memory should start out initialized to all zeros.

C2. A cache is a small but fast memory used to hold copies of recently accessed memory. The high speed of modern computers is surprisingly dependent upon the observation that if a program accesses a memory address once, it is likely to access it (and nearby addresses) again in the near future.

Caches typically work by dividing up main memory into logical groups called *lines* (typically 8 or 16 machine words), and then reading lines from and writing lines to main memory. (It is often faster to read a whole line at once than to to read an equal number of nonadjacent words). Whenever the CPU reads (or writes, in some caches) a particular address, the entire line containing that address is first moved to the cache.

Caches are small, so they cannot hold all the lines in main memory. A *direct-mapped* cache solves this problem by allocating space for l lines (one line's worth of space is called a slot) and declaring that the k^{th} goes in slot $k \bmod l$. (Thus when $l = 4$, line 1 and line 5 correspond to the same cache slot and cannot be in the cache simultaneously.) Each slot has a *tag* value that can be used to determine which of the possible lines is the slot (or that the slot is empty).[2]

Morgan and Terry have decided that the Noronic™ cache should also be *write back* and *write allocate*. Write allocate means that if an address being written is not already in the cache, the corresponding line should first be read into the cache (possibly evicting any previous occupant of the associated slot). Write back means that store instructions that write to memory only change the copy of memory in the cache (main memory holds the "old" value) until the line is forced out of the cache by another line coming into the cache; the evicted line is first written out to main memory.

There is some support in the *Cache* class for doing the slot/tag calculations, but the `read` and `write` member functions just call the corresponding operations in main memory—there isn't any caching going on.

Implement the caching scheme described above. Start by adding two data members,

2. In principle, the tag could just be the address of the first word in the line. But, since memory is divided up into non-overlapping lines, this address is always a multiple of the line size. We can thus save bits by making the tag be the address divided by the line size; real architectures always do this. The provided code includes ways to convert between tags and addresses, so the exact details shouldn't matter for your implementation.

```
indx_t*          tag_;     // numCacheslots_ tags
machine_word_t*  data_;    // (cacheslotSize_ × numCacheslots_) words
```

Given k cache slots and m words per slot, the former should point to an address of tags of size k, and the latter should be the address of an array of words, of length km. Modify the one constructor and the destructor to create and destroy these dynamically-allocated arrays.

Once you have implemented these arrays, you can uncomment the declaration and definition of the slotToAddr member function, and the commented out code in the dump member function.

C3. Next, implement the private member functions

```
void initSlot(indx_t slot);
size_t releaseSlot(indx_t slot);
size_t fetchLine(addr_t addr);
size_t ensureCached(addr_t addr);
```

where initSlot sets the tag for the slot to INVALID_TAG and sets the contents to zero, releaseLine ensures the given slot is free (by writing its contents to main memory if full), fetchLine puts the line containing the given address into the appropriate slot, and ensureCached makes sure that a given memory location is in the cache, calling fetchLine if needed but also calling releaseSlot if the desired slot is already full.

The last three functions all return the number of cycles required to do all necessary reads and writes to main memory.

C4. Modify *Cache*'s read and write member functions to use cached data and not directly refer to main memory. You'll probably want to thoroughly test the simulator at this point.

C5. It's impractical to always write cache lines back to main memory when they are evicted. Many lines (especially those containing code) are never modified between the time they enter and leave the cache, so the write is pointless. Real hardware handles this situation by adding *dirty flags*, a flag for each slot recording whether the line was written to since it was read from main memory. Modify the simulated cache so that it maintains a dirty flag for each slot, and avoids writing to main memory whenever possible.

Note that the dirty flags are *per line* and not per word—the entire line is always written back if any word in it was written to.

Extend the implementation of dump so that it puts a * next to dirty slots.

C6. Each time *Cache*::instr is invoked, the simulator must translate the bits in the machine word that specifies the current instruction into an *Instruction* object that can be executed. If code in the cache is repeatedly executed (e.g., a loop),

then this process can be inefficient. One way to speed up the simulator is to store the *Instruction* object the first time that address is translated, and then repeatedly return the same *Instruction* every time, until the code is overwritten or leaves the cache. (Real processors actually do tricks like this, too.) Implement this optimization by adding a data member *Instruction*`**` `instr_` that references a dynamically allocated array of pointers to *Instruction*s, an array with the same number of elements as `data_`. Modify the constructor(s) and destructor, and any other *Cache* methods as appropriate, so that repeated calls to `instr` on the same address will immediately return the same object—until this address is evicted from the cache.

Extend the implementation of `dump` so that it puts a + next to words stored in the cache that have an associated *Instruction* object.

C7. For the same amount of money, the following configurations are possible:

(a) 70 cycles to access an arbitrary memory location, 4 slots in the cache, and 4 words per line.

(b) 100 cycles to access an arbitrary memory location, 4 slots in the cache, but 8 words per line.

(c) 100 cycles to access an arbitrary memory location, 4 words per line, but 8 slots in the cache.

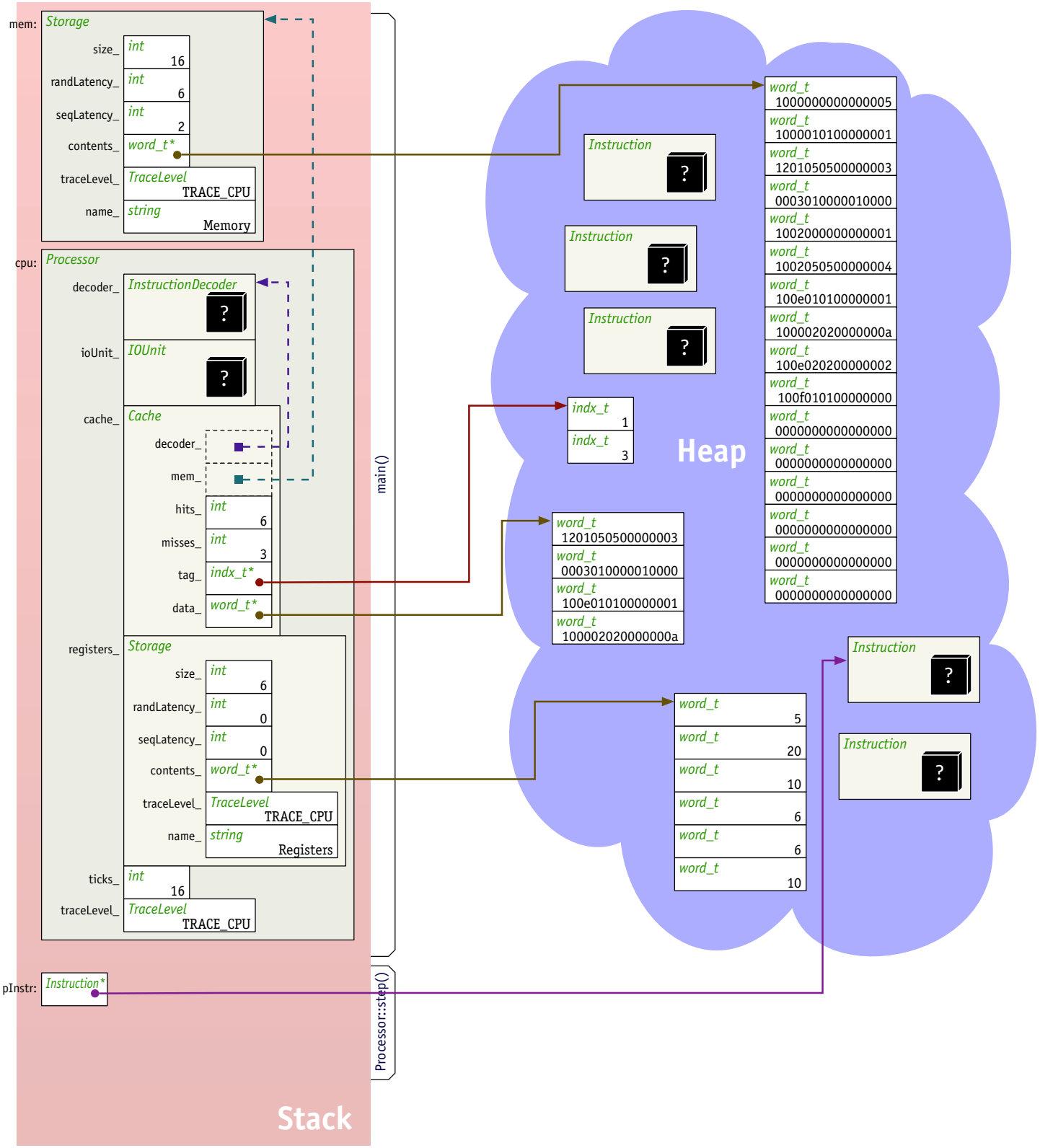
(all other values can be left at their defaults). Now that you have a working simulator, you can use it to help make a recommendation as to which configuration works best.

For testing purposes Terry and Morgan have provided two very similar programs, `matrix1` and `matrix2`. They first put the numbers 1 to N^2 into a square matrix (as provided, N is hardcoded as 13 in the first line) laid out in N^2 consecutive memory locations. Then a doubly-nested loop runs through all the rows and columns of the matrix, summing the elements. Finally the sum is printed. The difference between the two is that one sums the array by traversing the rows in order, hence going sequentially through memory; the other does the sum column by column, hence jumping around more in memory. To hold a matrix this large, you should set the memory to at least 200 words (`-m 200`).

In `HW3-Answers.txt`, describe of which these configurations would you recommend and why.

Useful Information

The diagram on the following page shows a possible state of memory in the simulator after coding stages C1 and C2 are complete. The black boxes with question marks indicate objects whose internals can be a "black box" to us. (And yes, there *is* a space leak at this stage.)


mem:
Storage
size_
int
16
randLatency_
int
6
seqLatency_
int
2
contents_
word_t*
traceLevel_
TraceLevel
TRACE_CPU
name_
string
Memory
cpu:
Processor
decoder_
InstructionDecoder
ioUnit_
IOUnit
cache_
Cache
decoder_
mem_
hits_
int
6
misses_
int
3
tag_
indx_t*
data_
word_t*
registers_
Storage
size_
int
6
randLatency_
int
0
seqLatency_
int
0
contents_
word_t*
traceLevel_
TraceLevel
TRACE_CPU
name_
string
Registers
ticks_
int
16
traceLevel_
TraceLevel
TRACE_CPU
main()
pInstr:
Instruction*
Processor::step()
Stack
Heap
Instruction
Instruction
Instruction
indx_t
1
indx_t
3
word_t
1000000000000005
word_t
1000010100000001
word_t
1201050500000003
word_t
0003010000010000
word_t
1002000000000001
word_t
1002050500000004
word_t
100e010100000001
word_t
100002020000000a
word_t
100e020200000002
word_t
100f010100000000
word_t
0000000000000000
word_t
0000000000000000
word_t
0000000000000000
word_t
0000000000000000
word_t
0000000000000000
word_t
0000000000000000
word_t
1201050500000003
word_t
0003010000010000
word_t
100e010100000001
word_t
100002020000000a
word_t
5
word_t
20
word_t
10
word_t
6
word_t
6
word_t
10
Instruction
Instruction