THE UNIVERSITY OF WESTERN AUSTRALIA

Department of Computer Science and Software Engineering

CITS2002 Systems Programming

Project 1 2021 - see also project 1 clarifications

The goal of this project is to emulate the execution of programs from a small, C-like programming language, on a stack-based machine with a write-back cache, and to report some metrics of the program's execution.

Successful completion of the project will enhance your understanding of some features of the C11 programming language, and develop your understanding of how simple programs can be executed on a general-purpose computer architecture.

Resources:	the coolc webpage	the *coolc.sh* shellscript	the *runcool.c* starting code	the cool language syntax	the stack-based machine instruction set

The *cool* programming language

The *cool* programming language (not a real programming language that you'll find on the web!) is a very small programming language with a C-like syntax. It is designed to be easily understood by C programmers (who will have no difficulty relating it to C's appearance and execution) so that attention may focus on the execution of the programs. Its syntax is presented here.

Here is a quick summary of *cool*'s (lack of) features:

- programs are written in text files, whose filename ends in the *.cool* extension.
- each self-contained program is written in a single source-file, and cannot access any external libraries.
- all programs must have a single '**int** main(**void**)' function, which receives no parameters, and returns an *integer* result providing the program's exit status.
- program termination must be through *main()*, as there is no *exit()* function.
- all variables (globals, parameters, locals) and expressions are of type *integer*.
- all variables are scalar variables; there are no array variables.
- by default, all variables are initialised to *zero*, or may be initialised to an *integer* constant.
- there is no *bool* datatype, but non-zero expressions are considered 'true' in evaluating *if* and *while* statements.
- all functions are of type *void*, or return an *integer*.
- all identifiers must be defined before their use (no forward declarations or mutually-recursive functions).
- to aid debugging, a single *print* statement prints either an *integer* expression or a character string (the only place that strings exist).

Compiling and running *cool* programs

Programs are compiled by the *coolc* compiler, which translates a *.cool* program to an 'executable' format, whose filename ends in the *.coolexe* extension. The compiler will only be available via a webpage while the project is running (details follow), but its source code (2410 lines, so far) will be made available *after* the project. This approach allows bugs to be corrected for *all* users, avoids the problems of providing multi-platform support, and avoids compiler versioning and distribution problems.

The *coolc* compiler generates code for a (ficticious) stack-based machine. Code generation is performed in a single-pass without the need for an intermediate assembly-language file (although *coolc* displays a 'fake' assembly-language file on its webpage). *cool* programs may be compiled in two (related) ways:

- The coolc webpage allows you to upload a text file containing your *cool* program, to be compiled on the web-server. Your source code will be displayed, together with any (at most one) syntax error. If your program contains no syntax errors, a (pseudo) assembly language version of your program will also be displayed, its instructions and data written to a *coolexe* file, which may be downloaded to your computer.

- The coolc shellscript may be downloaded to your computer and executed to send a text file containing your *cool* program to the web-server (described above). If the uploaded file contains a syntax error, a report will be returned to your computer. If your program contains no syntax errors, its instructions and data will be written to a *coolexe* file, which is downloaded to your computer.

```
prompt>  ./coolc.sh  myprogram.cool
```

Once you have a *coolexe* file on your computer, using either of the two mechanisms described above, you can emulate the execution of the program using **your** program, **your** project, named *runcool*:

```
prompt>  ./runcool  myprogram.coolexe
```

The stack-based computer architecture

The target computer is a stack-based machine, with no general-purpose registers. The stack is used to hold temporary results during arithmetic evaluation, and memory addresses to manage the flow of each program's execution (described later).

Control registers

Three specific on-CPU control registers control the execution of programs:

- PC - the *program counter*, holds the memory address of the *next* instruction to fetched from the computer's RAM.
- SP - the *stack pointer* points to the word on the *top* of the computer's stack.
- FP - the *frame pointer* points to the current function's activation frame of the computer's stack (described later).

The on-CPU control registers are modified in response to each program's instructions. For example, an instruction requiring an integer to be pushed onto the stack, results in SP being modified. Similarly, a function call results in the PC's value being pushed onto the stack, and the PC modified to point to the first instruction of the called function.

16-bit words

The computer on which *cool* programs execute can be described as an (aging) 16-bit computer [historic reference for those interested].

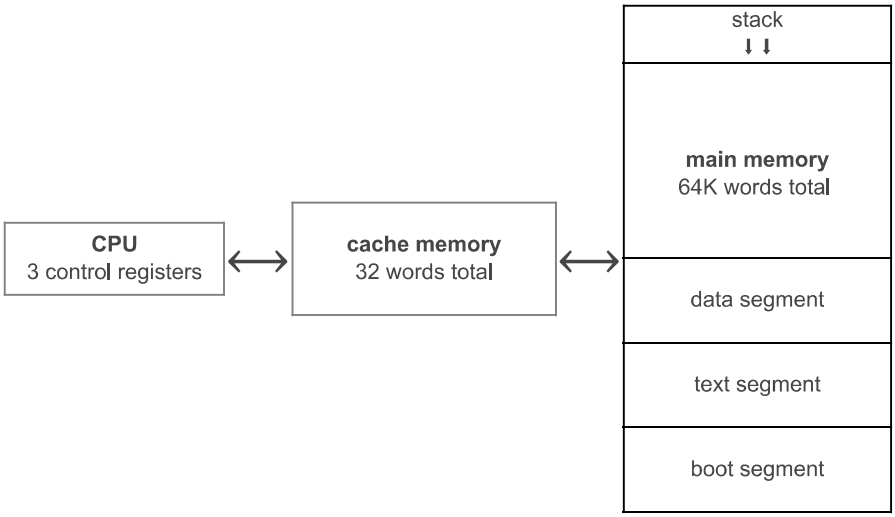
- a pair of 2 successive bytes are termed a (16-bit) *word*
- each *memory address* is 16-bits wide, meaning that there are 2^{16} distinct addresses.
- *memory addresses* may be considered as *unsigned* integers, ranging from 0 to 65535.
- memory is *word addressable*, meaning that each address refers to one of 2^{16} possible words of memory (or 128KB in total). The on-CPU registers - PC, SP, and FP - each hold one address, and are thus 16-bits wide.
- the only supported datatype is the *signed integer*, ranging from -32,768 to 32,767.
- conveniently, and by design, each address and each integer occupies the same sized unit of memory - one word.

Main memory and cache memory

Our stack-based machine has a 3-stage *memory hierarchy*. During a program's execution, requests to read and write memory are made by the CPU, and the requests and data 'flow' between CPU, cache-memory, and main-memory.

1. The CPU contains just three very fast on-CPU control registers, the PC, SP, and FP, but these cannot be used as general-purpose memory locations to store arbitrary values. These registers are only changed through the interpretation of a program's instructions.
2. The cache-memory is a relatively fast bank of memory (typically 8-20x faster than main-memory, but not as fast as the CPU's registers), and stores a total of just 32 (2^5) words.
3. The main-memory is a relatively slow 'bank' of memory, is used to store the program's instructions, data, and runtime stack, and stores a total of 64K (2^{16}) words,

Different computer architectures have varying sized memory hierarchies - a typical laptop computer may have 8GB of main-memory, 16MB of cache-memory, and 12+ on-CPU general purpose registers. If the instructions and data we want are in the fastest form of available memory, our programs will run faster. However, faster memory costs more money (built using different memory technologies) and, so, we typically cannot store all of our instructions and data in the fastest memory. As in the following image, cache-memory lies (logically) between the CPU and main-memory and, so, all requests to read instructions and to read or write data words are seen by the cache.



Because cache is much smaller than main-memory, each cache location must remember both its data and the main-memory location to which that data 'belongs'. When a word from main-memory is copied *through* the cache, that word's address is always 'mirrored' in the same cache location. There is only one single cache location for each main-memory location, but many potential main-memory locations for each cache location. Thus, each cache location must remember which main-memory location it is caching.

- Each word *read* from main-memory is retained by the cache and, if required again, is quickly taken from cache without 'asking' the main-memory for another (identical) copy. When the required data is *available* in the cache, we have a **cache-hit**. However, if the required data is *not available* in the cache, it must first be fetched from main-memory. This is termed a **cache-miss**.
- When the CPU *writes* data words, the cache has two potential courses of action:

  1. it could immediately write the word to the main-memory. As no other components can modify main-memory, the cache and main-memory are always 'in sync'. This is termed a **write-through policy**. Simple.

  2. alternatively, the cache could remember the new word, but *not* immediately copy it to main-memory. The cache and main-memory are now 'out of sync'. Any future *read* request for the same word can be satified by the cache. However, if the requested word is *not* in cache, it must be copied from the main-memory on its way to the CPU. As the new word requires the same cache location as the data that was *not* previously written to main-memory, that *delayed write* to main-memory must now be be performed first. For each word of cache, we need to remember whether is correctly reflects what is in main-memory - whether the cache's copy is *clean* or *dirty*. Dirty data needs writing to main-memory before its cache location can be re-used to store new data from main memory. This is termed a **write-back policy**.

Phew. **For this project, you must implement a write-back cache.**

You are also required to calculate (count) and report four basic statistics to measure the effectiveness of the cache:

- the number of main-memory reads,
- the number of main-memory writes,
- the number of cache-memory hits, and
- the number of cache-memory misses.

The instruction set

cool programs consist of a mixture of *instructions*, *addresses*, and *data* (integers). The computer's instruction set is presented here. (At present) there are only 17 instructions. Each instruction has a unique (integer) value (0..16) and each instruction requires one or two words of the computer's memory:

- Some instructions, such as *'add'*, require no additional information (here, because the values to be added reside on the stack). The very next word after an *'add'* instruction is the next instruction to be executed. *'add'* is an example of a 0-operand instruction, and requires one word in total.
- Some instructions, such as *'call'* require additional information. The very next word after a *'call'* instruction is the memory address of the first instruction of the function to be called. *'call'* is an example of a 1-operand instruction, and requires two words in total.

Calling and returning from functions

The implementation of functions is one of the more difficult concepts to understand, because:

- functions need to access their parameters and local variables,
- functions need to return to wherever they were called from,
- functions can call other functions (even recursively), and
- functions can return values to their caller.

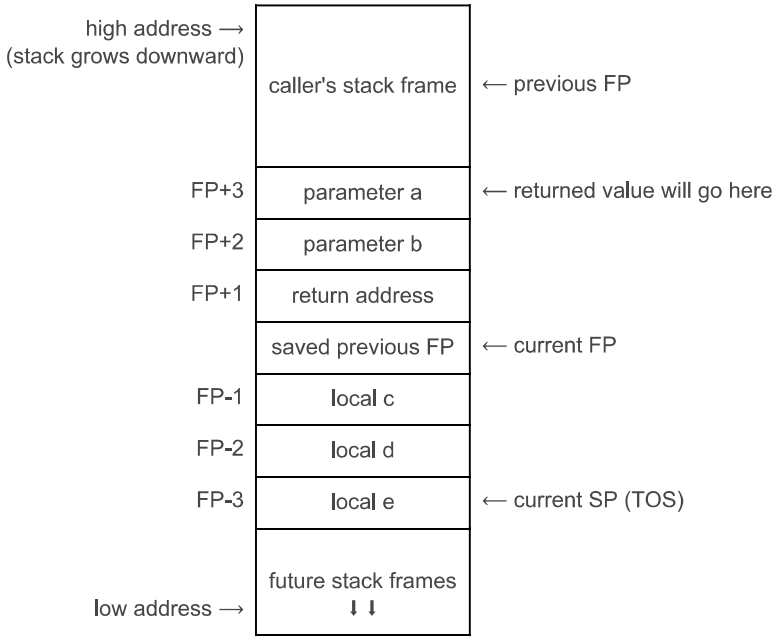
Each function employs memory on the stack to hold incoming parameters and local variables. This region of memory is called a *stack-frame* and resides on the stack. An on-CPU control register, FP (*frame-pointer*), allows us to locate the current function's frame. The code to access parameters and local variables while a function is executing is generated in terms of *offsets* to FP. Parameters appear 'above' FP (at higher addresses, positive offsets), and local variables appear 'below' FP (at lower addresses, negative offsets). The stack pointer, SP, may change during the execution of a function as values are pushed or popped off the stack (such as pushing parameters in preparation for calling another function). However, the FP doesn't change throughout the execution of a function.

Consider the following *cool* function, and the modifications to the on-CPU control registers and the stack when *myfunc()* is called:

1. the function wishing to call *myfunc()* passes 2 actual parameters by pushing them onto the stack (light green)
2. the PC now points to the *call* instruction, which is followed by the address of the first instruction of *myfunc()*.
3. so that we know where to return to, the address immediately following these two words (the address of the next instruction) is saved onto the stack - the 'return address' (pink).
4. the current value of FP (soon to be the previous value of FP) is saved onto the stack (pink), and the current value of SP, is copied into FP (to become *myfunc()'s* frame-pointer).
5. space for *myfunc()'s* 3 local variables (purple) is allocated by pushing their initialised values onto the stack.
6. execution may now contine from the first 'real' instruction of myfunc(), knowing that we can locate each of the parameters and local variables by their offsets from the current FP.
7. we can return the function's return value, by effectively pushing the returned value to the TOS of the caller's stack frame by overwriting the first element of *myfunc()'s* stack frame, (parameter a, at FP+3 in *this example*), and we can restore the FP to the correct value in the caller's stack frame.

```
int myfunc(int a, int b)
{
    int c = 4;
    int d;
    int e = 9;

    // first 'real' instruction of myfunc()
    // ...
    return c + d + e;
}
```



When returning from *myfunc()*, the on-CPU control registers need restoring to their correct values so that execution may continue within the function that called *myfunc()*. In addition, the function's returned value (which will be on its TOS) needs leaving on the TOS of the calling function - to the first location of the current frame. In the diagram above, the returned value will be copied to the same stack location as *parameter a*. Careful - if the function has *no* parameters, the returned value will be copied to the same stack location as the return address.

Project requirements

- You are required to develop and test a program, named *runcool*, that emulates the execution of *cool* programs, by interpreting the contents of a *coolexe* file running on the stack-based machine employing a write-back cache policy, as described in this project description.
- Your program should produce a number of execution statistics, each printed one-per-line, and preceded by a '@'. For example:

```
main calls function1
function1 calls function2
back in function1
back in main
my calculated result = 238
@number-of-main-memory-reads  1044
@number-of-main-memory-writes  106
@number-of-cache-memory-hits  3316
@number-of-cache-memory-misses  408
```

  During the project marking, all lines other than those presenting the required statistics, or printed by the *cool* program itself, will be ignored. Any statistic not reported will be assumed to provide the value of zero (which will never be correct).

- Your program must, itself, exit with the integer value left on the TOS by the *main()* function of any *cool* program.
- Your project must be written in C11 in a single source-code file named **runcool.c**
  This is the only file you should submit for marking.
  Your submission will be compiled with the C compiler arguments **-std=c11 -Wall -Werror**, and marks will not be awarded if your submission does not compile.
- Your program, named **runcool**, must accept exactly one command-line argument providing the name of a *coolexe* file:

```
prompt>  ./runcool  program.coolexe
```

- You should start by reading, understanding, and then extending **the starting code**. **DO NOT** modify any of the provided constants near the top of the file (they will be used during marking).
- You may use any functions from the standard C11 library but must not employ any 3rd-party code or libraries to complete your project. If in doubt, please ask.

Assessment

The project is due **11:59pm Friday 17th September (end of week 7).** and is worth **25% of your final mark** for CITS2002. It will be marked out of 50.

The project may be completed **individually or in teams of two** (but not teams of three). The choice of project partners is up to you - you will not be automatically assigned a project partner.

You are **strongly** advised to work with another student who is around the same level of understanding and motivation as yourself. This will enable you to discuss your initial design together, and to assist each other to develop and debug your joint solution. Work together - do not attempt to split the project into two equal parts, and then plan to meet near the deadline to join your parts together.

25 of the possible 50 marks will come from the correctness of your solution (**automated marking**). The remaining 25 marks will come from assessing your programming style, including your use of meaningful comments; well chosen identifier names; appropriate choice of basic data-structures, data-types and functions; and appropriate choice of control-flow constructs (**manual marking**).

During the marking, attention will obviously be given to the correctness of your solution. However, a correct and efficient solution should not be considered as the perfect, nor necessarily desirable, form of solution. Preference will be given to well presented, well documented solutions that use the appropriate features of the language to complete tasks in an easy to understand and easy to follow manner. That is, do not expect to receive full marks for your project simply because it works correctly. Remember, a computer program should not only convey a message to the computer, but also to other human programmers.

Submission requirements

1. The deadline for the project is **11:59pm Friday 17th September (end of week 7)**.

2. **You must submit your project electronically using *cssubmit*.**

   The *cssubmit* program will display a receipt of your submission. You should print and retain this receipt in case of any dispute. Note also that *cssubmit* does not archive submissions and will simply overwrite any previous submission with your latest submission.

3. Your submission's C11 source file **must** contain C11 block comment:

   ```
   //  CITS2002 Project 1 2021
   //  Name(s):             student-name1   (, student-name2)
   //  Student number(s):   student-number1 (, student-number2)
   ```

4. If working as a team, only one team member should make the team's submission.

5. Your submission will be examined, compiled, and run on a contemporary **Linux system** (such as Ubuntu). Your submission should work as expected on this platform. While you may develop your project on other computers, excuses such as *"it worked at home, just not when you tested it!"* will not be accepted.

6. This project is subject to UWA's Policy on Assessment - particularly §5.3 *Principles of submission and penalty for late submission*, and Policy on Academic Conduct. In accordance with these policies, you may *discuss* with other students the general principles required to understand this project, but the work you submit must be the result of your own team's efforts. All projects will be compared using software that detects significant similarities between source code files. Students suspected of plagiarism will be interviewed and will be required to demonstrate their full understanding of their project submission.

Good luck!

Amitava Datta & Chris McDonald.