CPSC 213, Summer 2007— Midterm — Solutions

Date: May 30, 2007; Instructor: Mike Feeley

1. **(12 marks)** Short Answers.

1a. What is wrong with this explanation of how a computer accesses a variable (lets call it A): "The computer asks the memory to give it the value of the variable named A."?

The memory does not know what a variable is and it doesn't know anything about variable names. The only language the memory understands is "give me the value stored at address x" or "write the value v at address x". Only the compiler knows about variables and their type. The compiler generates machine code that access variables by specifying their address.

1b. Consider the C global declaration: `int a[10]`. Does the compiler know the address of a[4]? If so, how does it calculate it?

Yes. Adds 4×4 to the base address of the array. It knows the base address because the array is static and thus the compiler allocates the memory for it and so knows where it put it.

1c. What is *pc-relative* addressing and why is it used?

It names a memory location using a small number added to or subtracted from the current value of the program counter. In the Gold ISA it is used for branch instructions, which is typical. The reason it is used so that nearby branches can be encoded by a two-byte instruction instead of using 6 bytes to completely specify the target memory address. Typically, all of the control flow changes within a procedure are to small offsets from the address of the branch instruction and thus all of these can be implemented using branches.

1d. Why must the Java compiler use a double indirect (or indirect) jump (i.e., Gold `croo`, `droo`, or `eri-`) to implement certain method invocations while the C compiler can use a direct jump to implement all procedure calls (i.e., `b--- aaaaaaaa`)?

Because, for instance methods, the Java statement that invokes a method does not determine the code address of the method that is invoked. The problem is that a Java variable can store references to objects of different type as long as they implement or extend the type of the variable. And, the invoked method's address is determined by the type of the object, not the type of the variable. This address is stored in memory at an offset from the object from the object. And so, invoking the method requires a jump that chooses its target address from memory; i.e., the double indirect jump. A single indirect jump can also be used if the target address is first loaded from memory into a register using a load instruction.

2. **(6 marks)** This program prints out the initial value of two variables `global` and `local`.

```
int global;
int main () {
    int local;
    printf ("global=%d, local=%d, k=%d\n", global, local);
}
```

When executed, it outputs `global=0, local=-1877226936` (and in fact the value of `local` appears to be random). Explain why these two variables have different initial values. Be sure to indicate where in memory the variables are stored and how their storage location impacts their initial value. State whether you think there is a good reason they have different initial values and if so clearly explain what that reason is.

The variable `global` is statically allocated by the compiler and thus the compiler can statically assign it an initial value. The variable `local` is dynamically allocated on the stack when `main ()` is called. Allocating a stack frame requires only moving the stack pointer to make room for the frame. Thus the initial value of a local variable is whatever happens to be in memory at its particular offset from the beginning of its frame. While it would be possible to give local variables an initial value, doing so requires executing instructions that do this explicitly, which has a runtime cost. Since there is a runtime cost to this initialization, but not for the initialization of globals, the better choices is to leave it to the application to decide on the initial value of variables. For this reason, Java does not assign local variables an initial value, but it also does not allow a program to read a local variable until the program assigns it a value.

3. **(6 marks)** Consider a language, call it D, that is exactly like C, except that it allows programers to declare static variables whose size is determined at runtime. In the example below, `a` is such a variable; its size is determined at runtime by the value of `i` passed in to `foo`.

```
void foo (int i) {
    int a[i];
    int j;
}
```

Your job is to assess the impact of this new feature on the machine code generated by the compiler. There are two important things that change about this machine code. What are they? Explain.

Two things that used to be statically determined no longer are. The first is the size of the stack frame. The code the compiler generates to allocate the stack frame must thus read the value of the argument, i, first and then use this value to compute the number of bytes to subtract from the stack pointer to make the frame. In C and Java, the compiler just generates code that subtracts a constant from the stack pointer. The same issue exists for de-allocation of the from when the procedure returns. The second issue is that the compiler no longer knows the stack offset of the variable j, because its position is determined dynamically by the size of the array. And thus, when generating code to access j, the compiler needs to read the value of i and it can not use the base-plus-static-displacement addressing mode.

4. **(10 marks)** Consider the following Java class.

```
public class Foo {
    static int i;
    int j;
    static void foo (int k) {
        int l;

        i=0;
        j=0;
        k=0;
        l=0;
    }
}
```

Give the Gold Machine instruction(s) for assigning each of the variables `i` through `l` the value 0.

Assume 0 is stored in `r0` and that `r5` stores the stack pointer and `r7` stores the address of the object. Also assume that any static variables allocated by the compiler start at address 0x1000. Clearly state any other assumptions you make.

Your answer should be numeric Gold machine instructions. Include a comment next to each instruction explaining what it does (the disassembler-style comment is fine).

4a. The code for `i=0`:

```
0100 00000100   # ld $0x100, r1
3001           # st r0, 0x0(r1)
```

4b. The code for `j=0`:

```
3007 # st r0, 0x0(r7)
```

4c. The code for `k=0`:

```
3015 # st r0, 0x4(r5)
```

4d. The code for `l=0`:

```
3005 # st r0, 0x0(r5)
```

5. **(12 marks)** Here is a Gold ISA implementation of a procedure that takes two arguments and returns a value. Your gold in this question is to figure out what this procedure does. Don't panic. You can do this if you first place helpful comments next to each line of the program and then use those comments to translate the program into C-like pseudo code. The procedure has a loop in it and it computes a simple, useful function on its inputs.

```
00000100: 6605          deca r5              # create stack frame
00000102: 6605          deca r5              # for 2 local variables
00000104: 0000 00000000 ld  $0x0, r0         # r0 = 0
0000010a: 3005          st  r0, 0x0(r5)      # local0 = 0
0000010c: 3015          st  r0, 0x4(r5)      # local1 = 0
0000010e: 1050          ld  0x0(r5), r0      # r0 = local0
00000110: 1351          ld  0xc(r5), r1      # r1 = arg1
00000112: 6701          not r1               # r1 = !arg1
00000114: 6301          inc r1               # r1 = -arg1
00000116: 6101          add r0, r1           # r1 = local0-arg1
00000118: 9109          beq  r1, 0x12a       # if (local0<arg1) goto 12a
0000011a: 1251          ld  0x8(r5), r1      # r1 = arg0
0000011c: 2101          ld  (r0,r1,4), r1    # r1 = arg0[local0]
0000011e: 1152          ld  0x4(r5), r2      # r2 = local1
00000120: 6112          add r1, r2           # r1 = arg0[local0]+local1
00000122: 3215          st  r2, 0x4(r5)      # local1 += arg0[local0]
00000124: 6300          inc r0               # r0 = local0+1
00000126: 3005          st  r0, 0x0(r5)      # local0++
00000128: 80f3          br  0x10e            # goto 10e
0000012a: 1150          ld  0x4(r5), r0      # r0 = local1
0000012c: 6405          inca r5                 # teardown stack frame
0000012e: 6405          inca r5
00000130: c600          jmp 0x0(r6)          # return local1
```

5a. Place useful comments above where you think necessary to understand the program (and for part marks). Don't just say things like "adds four to register 5". Either say nothing or say something a little more useful like "discards top word from stack".

```
int sum (int* a, int aLength) {
    int i;
    int s;
    s = 0;
    for (i=0; i<aLength; i++)
        s += a[i];
    return s;
}
```

5b. Translate the program into C-like pseudo code and explain in English (briefly) what function the program computes.

6. **(4 marks) BLUE TEAM ONLY**
Your instruction set does not include indexed load and store instructions such as Gold `2isd` and `4sid`. Give one advantage and one disadvantage of this design choice. Give an example of a C or Java language feature (or part of one) that requires more Blue machine instructions than Gold, in light of this choice; and explain why.

An advantage is that your ISA has fewer instructions and thus your hardware implementation will be simpler (and thus cheaper to design/debug etc); this can be a huge advantage. A disadvantage is that your ISA requires more instructions to implement memory access where the base and index are determined dynamically. Accessing elements of a dynamic array using a variable as the index, is a common example. In Gold, you can place the index in one register and the base address in another and then issue the load or store with one instruction. In Blue, you will need to load the index and base and then (1) multiply the index by four (e.g., by shifting left 2 bits) and (2) add this value to the base. You can now use your base+displacement instruction with displacement 0. Two extra instructions compared to Gold. Since accessing dynamic arrays using dynamic index variables is a common programming idiom, it is likely best to include both the indexed version of the instructions in your ISA.

6. **(4 marks) GREEN TEAM ONLY**
Your instruction set has five instructions that read or write memory, but does not include base plus displacement instructions such as Gold `1osd` and `3sod`. What was the tradeoff you made? Give one advantage and one disadvantage of this design choice. Give an example of a C or Java language features that requires more Green machine instructions than Gold, in light of this choice.

Perhaps you decided to include a number of load/store variants such as to store immediate values to memory and then realized you were running out of op codes (or design/implementation budget, if you were really building this) and thus decided not to include the base-displacement instructions. An advantage is that you can store an immediate into memory with one instruction; the Gold ISA requires two. A disadvantage is that your ISA requires more instructions to implement memory access where the base is dynamic but the displacement is known statically. For example, local variables and instance variables are accessed in this manner. In Green, to access a local variable at offset 4 from the beginning of the stack frame, it is necessary to first load the number 1 into a register and to then use the indexed load using this register as the index (its 1, because Gold multiplies the index by four). In Gold, this step is not necessary since the base+displacement load/store specifies a base register and a hardcoded, immediate, offset, in this case 1.

6. **(4 marks) MAUVE TEAM ONLY**
Your instruction set does not include base-plus-displacement load and store instructions such as Gold `1osd` and `3sod`. Give one advantage and one disadvantage of this design choice. Give an example of a C or Java language feature that requires more Mauve machine instructions than Gold, in light of this choice.

Same as Green, though your advantage is that you simply have fewer instructions to implement, like Blue. Your disadvantage is the same as Green's.

6. **(4 marks) TERRACOTTA TEAM ONLY**
In your instruction set both jumps and branches can be conditional, but your branch instructions use a 4-bit pc offset. In the Gold machine, only branches can be conditional, but these instructions have an 8-bit pc offset. What design tradeoff did you make (think about why you have only 4-bit offsets)? Give one advantage and one disadvantage of your choice. Give an example of something Terracotta can due in fewer instructions than Gold and vice versa.

You added extra instructions — the conditional jumps — but ran out of op-code room and so had to use one opcode for all conditional branches. As a result, you were left with only 4-bits in the instruction to store the PC offset. In Gold, on the other hand, `beq` and `bgt` are different opcodes and thus Gold has 8-bits for the PC offset. This means that Terracotta can only use pc-relative addressing to go forward 7 instructions or backward 8. Gold can go forward 127 instructions and backward 128. So, your advantage is that you can do a conditional jump in fewer instructions than Gold. Gold would have to combine conditional branches and an unconditional jump to do the same thing. On the other hand, your disadvantage is that compilers for your ISA will have to use jumps more often than branches, because of the limited range of your pc-relative addressing. Since jumps are 6 bytes and branches 2, anytime you have to use a jump instead of a branch, there a reasonably large performance hit. Furthermore, it turns out that while many localized gotos are conditional (e.g., for loops and for if-then-else), most distant gotos are unconditional (e.g., for procedure calls). Therefore if you can get your pc-relative addressing to cover the code for one method, then you likely really don't need conditional jumps in the ISA.