ECE 4750 PSET 1

Tim Yao (ty252)

Sept 24, 2015

1 Comparing CISC, RISC, and Stack ISAs

1.a x86 CISC ISA

The x86 program contains 3+2+3+1+2+2=13 bytes.
If the size = 10, the number of bytes of instructions that need to be fetched is 3+2+2+2+10*(3+1+2+2)=89 bytes. The xor and jmp instructions are both executed once at the beginning of the program. The instructions cmp and jl are also executed at the beginning of the program, before the actual looping starts. Every loop contains the two inc instructions, cmp, and jl.
Assuming 32-bit data values, a total of 4*10 = 40 bytes need to be both loaded and stored to the data memory.

1.b MIPS32 RISC ISA

```
1         sll r5, r5, 2      // Multiply the size by 4 due to byte addressed memory
2         jmp L1             // Jump to L1 to do the initial size != 0 check.
3         addi r12, r4, r5   // Calculate the absolute end location of the array in memory
4 loop:   addi r13, r13, 1   // Increment r13 by 1
5         sw r13, 0(r14)     // Store r13 back into the current spot in array
6         addi r4, r4, 4     // Increment the counter r14 by 4 (memory is byte addressed)
7 L1:     bne r12, r4, loop  // If r12 != r4, then go to "loop" and continue through the array
8         lw r13, 0(r4)      // Load the current item in the array into r13
```

There are two types of optimization in the code. The first takes advantage of the branch delay slots after the jump and branch instructions. The second restructures the looping mechanism to reduce the number of calculations and register usage. The addi instruction on line 3 takes advantage of the branch delay slot after the jump instruction on line 2. This is possible because the addi instruction needs to be executed regardless of the jump. The load word instruction on line 8 also takes advantage of the branch delay slot right after the branch instruction on line 7. The load word instruction is used for retrieving one element from the array and place it into the temporary register r13. Normally, this instruction is placed at the beginning of the loop, but because r13 is a temporary register, we can also load into r13 without any consequences (regardless whether the branch is taken or not). During normal looping (when bne is branching), the load word instruction saves a cycle each. When the loop is completed, that instruction will load an element beyond the scope of the array into r13. In this specific situation, this out of bounds load should not cause any problems, because r13 is a temporary register and we will not be using it after the loop has completed.
My MIPS32 program contains 4*8 = 32 bytes (8 instructions with instruction taking up 4 bytes).
A total of 4*(3+2+10*5) = 220 bytes of instructions need to be fetched if size is 10.
Assuming 32-bit data values, 4*10 + 4 = 44 bytes of data need to be loaded from memory. 4*10 = 40 bytes of data need to be stored to memory.

1.c Simple Stack-Based ISA

Contents of Stack on First Iteration	Access Stack Memory?	Label	Instruction
aptr; size			goto L1
aptr; size		loop:	swap
size; aptr	Y		dup
size; aptr; aptr	Y		dup
size; aptr; aptr; aptr			pushm
size; aptr; aptr; mem	Y		pushi(1)
size; aptr; aptr; mem; 1	Y		add
size; aptr; aptr; (mem+1)	Y		popm
size; aptr	Y		pushi(4)
size; aptr; 4	Y		add
size; (aptr+4)			swap
(aptr+4); size;	Y		pushi(-1)
(aptr+4); size; -1	Y		add
aptr; size	Y	L1:	dup
aptr; size; size	Y	loop:	bgtz loop

Figure 1: array_increment loop in stack based ISA
The final stack is (aptr+4); (size-1) after add on 3rd to last row

The stack based ISA program contains 5+1+5+1+1+1+1+2+1+1+2+1+1+2+1=26 bytes.
If size is 10, a total of 5+1+5+10*(1+1+1+1+2+1+1+2+1+1+2+1+1+5)=221 bytes of instructions need to be fetched.
Assuming 32-bit data values, a total of 4*10=40 bytes of data need to be both fetched from and stored to data memory. 11 instructions will result in some kind of access to the special stack memory. If size is 10, the special stack memory needs to be accessed a total of 2+10*11=112 times. If the microarchitecture implemented eight 32-bit registers for the stack, this program will never need to access the special stack memory. This is because the largest the stack will be during execution is 5 elements.

1.d Comparison of ISAs

The x86 ISA should give the smallest static code size and fewest dynamically fetched instruction bytes compared to the MIPS32 and stack based ISA. This is mainly due to it being a CISC and having a variable instruction length. In comparison to the MIPS32, which has all 4-byte instructions, the x86 ISA can reduce both the static and dynamic code size with instructions that are generally less than 4 bytes. In comparison to both the MIPS32 and the stack based ISA, x86 also uses a complex instruction set (CISC), which purpose is to allow a program to be written with the fewest lines of assembly as possible. While this increases the complexity of the microarchitecture and makes the ISA harder to work with, it does allow reduce the number of instructions is needed for a program. As already shown by the array_increment program, x86 uses the fewest instructions. While the stack based ISA also uses variable length instructions, the stack design is not well suited for most programs. The ability to only access the top of the stack requires extensive use of the swap instruction, which decreases the benefits of the variable length instructions.