ECE 4750 Computer Architecture
Topic 1: Processor Concepts

http://www.csl.cornell.edu/courses/ece4750
School of Electrical and Computer Engineering
Cornell University

revision: 2024-08-27-20-45

List of Problems

Problem 1. Understanding Assembly Code

In this problem, you will explore several unknown functions implemented using TinyRV1 assembly. For these problems, you can assume the standard RISC-V calling convention. Recall that this means that arguments are passed in registers `x11`–`x17`, the return value is stored to `x10`, and the return address is stored in `x1`. `x5`–`x7`, `x28`– are used as temporary registers. The instruction semantics for the TinyRV1 instruction set are included in an appendix for your reference. As in lecture, you can assume that any array lengths given as an input to the function are greater than zero.

Part 1.A Mystery Function #1

Consider the following unknown function. **Write a C function that clearly represents the functionality of this assembly sequence.**

```
loop:
 lw   x5,  0(x12)
 mul  x5,  x5, x15
 lw   x6,  0(x13)
 add  x6,  x6, x5
 sw   x6,  0(x11)
 addi x11, x11, 4
 addi x12, x12, 4
 addi x13, x13, 4
 addi x14, x14, -1
 bne  x14, x0, loop

 jr   x1
```

Part 1.B Mystery Function #2

Consider the following unknown function. **Write a C function that clearly represents the functionality of this assembly sequence.**

```
loop:
 lw    x5,  0(x13)
 slli  x5,  x5, 2
 add   x6,  x12, x5
 lw    x7,  0(x6)
 sw    x7,  0(x11)
 addi  x11, x11, 4
 addi  x13, x13, 4
 addi  x14, x14, -1
 bne   x14, x0, loop

 jr    x1
```

Part 1.C Mystery Function #3

Consider the following unknown function. **Write a C function that clearly represents the functionality of this assembly sequence.**

```
  addi  x10, x0, 0

 loop:
  lw    x6,  0(x12)
  bne   x6,  x0, foo
  jal   x0,  bar

 foo:
  sw    x6,  0(x11)
  addi  x11, x11, 4
  addi  x10, x10, 1

 bar:
  addi  x12, x12, 4
  addi  x13, x13, -1
  bne   x13, x0, loop

  jr    x1
```

Problem 2. Comparing CISC, RISC, and Stack ISAs

In this problem, your task is to compare three different ISAs: x86 is an extended accumulator, CISC architecture with variable-length instructions; MIPS32 is a load-store, RISC architecture with fixed-length instructions; and we will also look at a simple stack-based ISA with variable-length instructions.

Part 2.A x86 CISC ISA

Consider the following C code which takes an array pointer (`aptr`) and the number of elements (`size`) as inputs, and then adds one to each of the elements in the array.

```
void array_increment( int* aptr, int size )
{
  int i;
  for ( i = 0; i < size; i++ )
    aptr[i] = aptr[i] + 1;
}
```

The loop in the above C function might compile to the following assembly code. This assembly fragment is just for the loop; it excludes the code required for managing the stack and for returning from the function. On entry to this code, register `edx` contains `aptr` and register `ecx` contains `size`.

```
      xor eax, eax
      jmp L1
loop: inc [edx+eax*4]
      inc eax
L1:   cmp eax, ecx
      jl  loop
```

Spend some time understanding how the assembly code implements the C code. The meanings and instruction lengths of the instructions used above are given in Figure 1. A register specifier uses a `r` prefix, a register value is denoted as R[`specifier`] and a memory value is denoted as M[`address`].

Notice that there are two versions of the `inc` instruction: a register-register version and a memory-memory version with a more sophisticated addressing mode. The `jl` instruction implements a conditional jump by using the `SF` and `OF` condition flags. These condition flags are set by the instruction preceding the jump based on the result of the computation. Some instructions, such as the `cmp` instruction, perform a computation and set the condition flags, but do not return any result. The `OF` condition flag indicates overflow and is set if the result exceeds the positive or negative limit of the number range. The `SF` condition flag indicates the sign of the result and is set if the result is negative (less than zero). Thus the `jl` instruction implements a jump if less than operation by checking to see if $\texttt{SF} \neq \texttt{OF}$.

How many bytes are in the x86 program? How many bytes of instructions need to be fetched if `size` is 16? Assuming 32-bit data values, how many bytes of data need to be loaded from the data memory? How many bytes need to be stored to the data memory? Show your work.

Instruction	Operation	Length
`cmp rs1, rs2`	*temp* ← R[`rs1`] - R[`rs2`] `SF` ← sign bit of *temp* `OF` ← overflow of *temp*	2 bytes
`inc rt`	R[`rt`] ← R[`rt`] + 1	1 byte
`inc [rs1+rs2*imm]`	*temp* ← R[`rs1`] + (R[`rs2`] × `imm`) M[*temp*] ← M[*temp*] + 1	4 bytes
`jl label`	if (`SF` ≠ `OF`) jump to the address specified by `label`	2 bytes
`jmp label`	jump to address specified by `label`	2 bytes
`xor rt, rs1`	R[`rt`] ← R[`rt`] ⊕ R[`rs1`]	2 bytes

Figure 1: x86 ISA Subset

Label	x86 Instruction	Equivalent TinyRV2 Instruction Sequence
	`xor eax, eax`	
	`jmp L1`	
`loop:`	`inc [edx+eax*4]`	
	`inc eax`	
`L1:`	`cmp eax, ecx`	
	`jl loop`	

Figure 2: Direct Translation from x86 to RISC-V

Part 2.B RISC-V ISA with Direct Translation

Create a table similar to the one in Figure 2, and translate each of the x86 instructions into one or more TinyRV2 instructions. You should use the minimum number of instructions to translate each x86 instruction, but you should not do any optimization across x86 instructions. Ultimately, the direct translation of all of the x86 instructions should result in a TinyRV2 program that when executed would achieve the same result as the original x86 code. This means that you need to use the appropriate RISC-V registers in each translation so that the RISC-V program functions correctly. On entry to the RISC-V code, assume register `x10` contains `aptr` and register `x11` contains `size`. If necessary, use `x5`, `x6`, and `x7` for temporaries. Do not use any RISC-V pseudo-instructions. For example, do not use `li`, `la`, `j` since these pseudo-instructions can turn in a variable number of real instructions based on the context. We want to better understand the cycle-level execution of the code, and thus we need to focus on the real instructions that need to be executed.

How many bytes are in the TinyRV2 program using your direct translation? How many bytes of instructions need to be fetched if `size` is 16? Assuming 32-bit data values, how many bytes of data need to be loaded from the data memory? How many bytes need to be stored to the data memory? Show your work.

Part 2.C RISC-V ISA with Optimized Translation

Write an optimized TinyRV2 assembly program that implements the `array_increment` function from the previous section. In lecture, we discussed how we will often assume that the length of arrays is always greater than one. To be enable a fair comparison for this problem, we will not make this assumption. Your assembly code will need to verify that the size is greater than zero before starting to work on the input array. On entry to the RISC-V code, assume register `x10` contains `aptr` and register `x11` contains `size`. If necessary, use `x5`, `x6`, and `x7` for temporaries. You do not need to worry about the instructions required to return from the function (i.e., the `jr/jalr` instruction). As in the previous problem, you should not use pseudo-instructions.

Note that there are more efficient ways than simply translating each individual x86 instruction directly into TinyRV2 instructions. Try to optimize your code so that it minimizes register usage, static instructions, and/or the number of instructions fetched. You can assume the microarchitecture is fully bypassed. Obvious optimization approaches to consider are using software scheduling to avoid load-use stalls and more efficiently managing the loop control and array addressing. The arguments to the `array_increment` function are passed by value, which means that the values in `x10` and `x11` do not need to be preserved; you are free to overwrite these values. **Your solution should contain commented assembly code and an explanation of your optimizations.**

How many bytes are in your TinRV2 program? How many bytes of instructions need to be fetched if `size` is 16? You do not need to count instructions which are fetched but then later squashed! **Assuming 32-bit data values, how many bytes of data need to be loaded from the data memory? How many bytes need to be stored to the data memory? Show your work.**

Part 2.D Simple Stack-Based ISA

In a stack-based architecture, all instructions operate on an architecturally visible hardware stack. Some number of items at the top of the stack are kept in fast hardware registers, while the rest of the conceptually infinite stack is kept in a special portion of memory. Stack-based architectures were popular in the 1960's with the Burroughs B5000 computer serving as a classic example. This kind of architecture is convenient for expression evaluation, subroutine calls, recursion, and compiling for some important high-level languages such as ALGOL. Stack-based architectures fell out of favor in the late 1970's, but there has a been some recent interest owing to the stack-based nature of Java's intermediate bytecode representation. Figure 3 defines a simple stack-based ISA that we will use in this part. The semantics of each instruction are defined in terms of push and pop operations: pop v means pop the top of the stack and store it in variable v, and push v means push the variable v onto the stack. Note that v is not part of the architecture nor the microarchitecture – it is simply a notational construct to help define the instruction semantics. A memory value is denoted as M[*address*].

All values are 32-bit. In our simple stack-based ISA, only the `pushm` and `popm` instructions access memory; all other instructions remove zero, one, or two operands from the stack and replace them with the result (if there is one). The `swap` instruction is special since it essentially reads two values from the top of the stack and writes two values onto the top of the stack in one instruction. The opcodes are encoded in a single byte. Notice that in this ISA, `pushi` only handles an 8-bit immediate that is sign extended to 32 bits when written to the top of the stack. The instruction and memory addresses are four bytes.

Instruction	**Operation**	**Length**
`add`	pop *v0*; pop *v1*; push *v1* + *v0*	1 byte
`bgtz label`	pop *v*; if $v > 0$, jump to address specified by `label`; else, continue with next instruction	5 bytes
`dup`	pop *v*; push *v*; push *v*	1 byte
`goto label`	jump to address specified by `label`	5 bytes
`pushi imm`	push sext(`imm`)	2 byte
`pushm`	pop *addr*; push M[*addr*]	1 byte
`popm`	pop *v*; pop *addr*; M[*addr*] $\leftarrow$ *v*	1 byte
`sub`	pop *v0*; pop *v1*; push *v1* - *v0*	1 byte
`swap`	pop *v0*; pop *v1*; push *v0*; push *v1*	1 byte

Figure 3: Simple Stack-Based ISA

Contents of Stack on First Iteration	**Access Stack Memory?**	**Label**	**Instruction**
aptr; size			`goto L1`
aptr; size		`loop:`	`...`
			`add rest of instrs here`
			`...`
aptr; size	Y	`L1:`	`dup`
aptr; size; size	Y		`bgtz loop`

Figure 4: `array_increment` **Loop Implemented with Stack-Based ISA**

The microarchitecture assumed for this problem implements the top of the stack with two 32-bit registers and the remainder of the stack is stored in a special stack memory. We call these two 32-bit registers the top-of-stack (TOS) registers (the Burroughs B5000 also had two TOS registers). If there are two or more items on the stack and the program pushes another item on the stack, then the microarchitecture will take care of writing one item from the TOS registers to the special memory. If there are more than two items on the stack and the program pops one item from the stack, then the microarchitecture will take care of reading one item from the special memory into the TOS registers. To be more precise, assume an instruction pops n items ($0 < n$) at the beginning of its execution and pushes m items at the end of its execution. For our simple stack-based instruction set, $0 \leq n \leq 2$ and $0 \leq m \leq 2$ for all instructions. If $n < m$ and the final stack size is greater than two, the microarchitecture will need to write some number of items from the TOS registers to the special stack memory. If $n > m$ and the initial stack size is greater than two, the microarchitecture will need to read some number of items from the special stack memory into the TOS registers. If $n = m$, then there is no need to access the special stack memory regardless of the current stack size; in this case, the pushes and pops necessary to implement the instruction can simply be done by accessing the TOS registers. In other words, the `pushm`, `swap`, and `goto` instructions never need to access the special stack memory since $n = m$, while all other instructions may access the special stack memory depending on the current stack size.

Translate the `array_increment` loop from Part 2.A to our simple stack-based ISA. On entry to the code, assume that the top of the stack contains `size` and the second entry in the stack contains the `aptr`. To show your translation, **create a table like the one shown in Figure 4.** Explicitly track what is in the stack before the execution of each instruction, and note on which instructions the microarchitecture will need to perform some number of access to the stack memory. To get you started, a portion of the translation is already shown in Figure 4.

How many bytes are in the stack-based program? How many bytes of instructions need to be fetched if `size` is 16? Assuming 32-bit data values, how many bytes of data need to be loaded from the data memory? How many bytes need to be stored to the data memory? How many instructions will result in some kind of access to the special stack memory? If the microarchitecture used eight TOS registers, how many instructions would result in some kind of access to the special stack memory? Show your work.

Part 2.E Comparison of ISAs

Given the results from the first four parts as a starting point, **make a compelling argument for which ISA will result in smallest static code size and fewest dynamically fetched instruction bytes on a broader selection of common programs.** While you certainly should summarize the results from the first three parts, your analysis should not be purely based on these results. Consider how these initial results can be extrapolated to other programs. Do not factor in memory traffic or performance; your argument should be purely based on static code size and number of bytes of instructions that need to be fetched.

Appendix A: TinyRV1 Instruction Set

Assembly Syntax		**Semantics**
`add`	`rd, rs1, sr2`	R[`rd`] $\leftarrow$ R[`rs1`] + R[`rs2`]
`addi`	`rd, rs1, imm`	R[`rd`] $\leftarrow$ R[`rs1`] + sext(`imm`)
`mul`	`rd, rs1, rs2`	R[`rd`] $\leftarrow$ R[`rs1`] $\times$ R[`rs2`]
`lw`	`rd, imm(rs1)`	R[`rd`] $\leftarrow$ M[R[`rs1`] + sext(`imm`)]
`sw`	`rs2, imm(rs1)`	M[R[`rs1`] + sext(`imm`)] $\leftarrow$ R[`rs2`]
`jal`	`rd, imm`	R[`rd`] $\leftarrow$ PC + 4; PC $\leftarrow$ PC + sext(`imm`)
`jr`	`rs1`	PC $\leftarrow$ R[`rs1`]
`bne`	`rs1, rs2, imm`	if (R[`rs1`] != R[`rs2`]) PC $\leftarrow$ PC + sext(`imm`) else PC $\leftarrow$ PC + 4

R[`i`]: the value in register `i`
M[`ma`]: the value in memory at address `ma`
PC: program counter
sext: sign extend
assume all instructions include PC $\leftarrow$ PC + 4 unless otherwise specified