LSU EE 4720 Homework 2 Solution Due: 7 March 2003

Design a stack ISA with the following characteristics:

- Memory has a 64-bit address space and consists of 8-bit characters.
- The stack consists of 64-bit registers.
- The ISA uses 2's complement signed integers.
- Only add other data types as necessary.

The stack ISA must realize these goals:

- Small program size.
- Low energy consumption. (For here, assume energy consumption is proportional to dynamic instruction count.)
- Relatively simple implementation. Instructions should be no more complex than RISC instructions.

Design the instruction set based on the sample programs in the problems below and the following:

Arithmetic and Logical Instructions

They should read their source operands from the top of stack (top one or two items) and push their result on the top of stack. Arithmetic instructions cannot read memory and they cannot read beyond the top two stack elements. (That is, you can't add an element five registers down to one ten registers down. Instead use rearrangement instructions before the add.) Specify whether the arithmetic and logical instructions pop their source operands. One can have both versions of an instruction. For example, `add` might pop its two source operands off the stack while `addkeep` might leave the two operands:

```
# Stack:  26 3 2003
add
# Stack:  29 2003
addkeep
# Stack:   2032 29 2003
```

Remember that arithmetic and logical instructions cannot rearrange the stack and cannot access memory.

Immediates

Decide how immediates will be handled. There can be immediate versions of arithmetic instructions or one can have push immediate instructions. See the example below. Keep in mind that the register size is 64 bits.

```
# Stack: 123
addi 3   # An immediate add.
# Stack: 126
pushi 3
# Stack: 3 126
add
# Stack: 129
```

Load and Store Instructions

Memory is read only by load instructions which push the loaded item on the stack. Memory is written only by store instructions which get the data to store from the top of the stack. Determine which addressing modes are needed for loads and stores, and design instructions with those modes.

Stack Rearrangement Instructions

The stack rearrangement instructions change the order of items on the stack. Consider adding the following: `exch`, swaps the top two stack elements. `roll n j`, remove the top `j` stack elements and insert them starting after what was the `n`th stack element. (See the example.) Other stack rearrangement instructions are possible.

```
# stack 11 22 33 44 55 66
exch
# stack 22 11 33 44 55 66
roll 5 2
# stack 33 44 55 22 11 66
```

Control Transfer Instructions

Your ISA must have instructions to perform conditional branches, unconditional jumps, indirect jumps, and procedure calls. It must be possible to jump or make a procedure call to anywhere in the address space. (The only thing special the instruction used for a procedure call has to do is save a return address.) The branch instructions can (but do not have to) use a condition code register. No other registers can be used (other than those in the stack). Don't forget about the target address.

Problem 1: As specified below, describe your ISA and the design decisions used. (Don't completely solve this part until you have solved the other problems.)

(*a*) For each instruction used to solve the problems below or requested above, show the assembler syntax and the instruction's coding. The coding should show the opcode, immediate, and any other fields that are present. Don't forget the design goals. Also don't forget about control transfer targets.

There is no need to list a complete set of instructions, but for coding purposes assume their existence. (There must be a way of coding a complete set of instructions that realize the goals of this stack ISA.)

The solution to this part appears after the last problem.

(*b*) Determine the size of the stack. Specify instruction coding and implementation issues used to determine the size.

Stack size was set to 32 elements. The solution to Problem 3 used six stack elements (at most), if fewer were available additional instructions would be needed to move items from and to memory. Too many stack elements, say thousands, would be difficult to implement and a nightmare to program (or would go unused). Additional variations of the **index** and **roll** instructions would be needed to handle thousands of registers since one byte could not reference them all. A stack size of 32 was chosen to match the number of general purpose registers in RISC ISAs.

(*c*) Explain your decision on whether there are immediate versions of arithmetic instructions. (The alternative is instructions like `pushi`.)

A design goal was to include one-byte immediate arithmetic and other instructions. To do this only a few instructions could use immediates. This was limited to only a few arithmetic and logical instructions, the others could only get operands from the stack.

(*d*) Explain your selection of memory addressing modes. Also, pick an addressing mode that you did not use and explain why not.

The variable instruction size made it easy to include direct addressing (instruction holds entire memory address). Displacement addressing was included because it is so commonly used but the offset was limited to one byte to save opcodes. (RISC ISAs need larger offsets since they lack a direct addressing mode and so the offset is used as the second half of an address, the first being loaded into a register with an instruction like **lui**.) Indirect addressing was included to keep code size down. (That is, a **load.o.word 0** loads the same address as **loadr.sw** but it uses two bytes instead of one.)

Memory indirect, and postincrement addressing would have helped in reducing code size, they were not included since instructions could not exceed RISC-like complexity.

(*e*) Explain how other design decisions you have made help realize the goals of small program size, low energy, or simple implementation.

The five-bit-opcode, three-bit-operand format allowed many instructions to be coded in one byte, reducing program size. The use of a single 3-bit immediate field for many instructions simplifies implementation.

(*f*) Describe any design decision you made that involved a tradeoff between code size, energy, or implementation simplicity. (Pick any pair.) *The original question asked only about code size and energy.* If you didn't make such a decision make one up.

The ISA described above does not involve energy and code size tradeoffs, so here's a made-up decision. In ISA A every instruction has a one-byte opcode and any immediates must start in the second byte of an instruction. In ISA B there is a five-bit opcode and a three-bit immediate. With the exception of a **push** the three-bit immediate is the only kind of immediate an instruction can use (unlike the ISA described elsewhere in this homework). The **push** uses as many bytes as it needs for the immediate. Suppose 20% of dynamic instructions in ISA A could use the 3-bit immediate (if it were available), 10% require a one-byte immediate, and the remainder don't use immediates. The dynamic instruction count for ISA B would be 10% longer because of the added **push** instructions. On the other hand while those extra 10% instructions are two bytes each, the 20% of instructions that use a 3-bit immediate are 1 byte in ISA B but are two bytes in ISA A. Assuming the dynamic count is a reasonable predictor of the static code size, ISA B has smaller code size.

Problem 2: Re-write the following MIPS code in your stack ISA.

```
  lui $a0, %hi(array)       # High 16 bits of symbol array.
  ori $a0, $a0, %lo(array)  # Low 16 bits of symbol array.
  jal lookup                # The name of a routine.
  nop

# Push handle large immediates no need to use two instructions.
push.v array
jl lookup
```

Problem 3: Re-write the solution to Homework 1 in your stack ISA, use the template below. (Use your own solution or the one posted.)

```
lookup:
        # Call Arguments  (TOS is the top of the stack.)
        #
        # TOS:     Return address
        # TOS + 1: ADDR of first element of array.  Array holds 64-bit integers.
        # TOS + 2: Number of elements in array.
        # TOS + 3: TARGET, element to count.
        #
        # Return Value
        #
        # TOS: Number of times TARGET appears in the array starting at ADDR.

        # Solution Here
        #
        # [ ] Don't forget the return.


        # Solution

        # ra ptr size target
        push.0
        # count ra ptr size target
        rolls 2 5
        # ptr size target count ra
        rollu.3
        # size target ptr count ra
        sll.3
        index.2
        # ptr sizex8 target ptr count ra
        add
        # end target ptr count ra
        rolld.3
        # ptr end target count ra
        cmpk.eq
        b.1 DONE

LOOP:
        # ptr end target count ra
        index.0
        loadr.sw
        # data ptr end target count ra
        index.3
        # target data ptr end target count ra
        cmp.eq
        # target=data ptr end target count ra
        rolld.5
        # count target=data ptr end target ra
        add
        # count ptr end target ra
        rollu.4
```

```
# ptr end target count ra
addpower.3
cmpk.eq
b.1 LOOP

# ptr end target count ra
pop pop pop
# count ra
rollu.2
# ra count
j
```

Comments on Solutions

Most solutions to this assignment included substantially correct programs, however several common mistakes or less-than-optimal choices were made in the stack ISAs. The following are common mistakes:

Lack of one-byte instructions. Since program size is a goal frequently used instructions should take one byte, whenever possible. Some solutions omitted any one-byte instructions, increasing code size.

Lack of immediate arithmetic instructions. A design goal was to reduce program size, including the ones in the assignment. If there are no immediate arithmetic or logical instructions then whenever an immediate is needed a push instruction must also be included, adding to program size. A justification given in some solutions for omitting immediate arithmetic instructions is a reduction in complexity or instruction count. Though these would be reduced, it would be at the expense of code size and energy, two other design goals.

Lack of one-byte immediates. Many solutions had ISAs with a single immediate size, sometimes very large. Since the goal is small program size and since many instructions can use small immediates, there should be some instructions using one-byte immediates. Other instructions could use larger immediates. (There is no reason why there should be a single immediate size.)

Lack of a 64-bit immediate. Since the register size is 64-bits there should be an instruction that can load a 64-bit constant, for example, `pushi 0xfedcba9876543210`.

Inclusion of `lui`-like instructions. Many solutions included instructions similar to MIPS load-upper immediate. Such instructions make sense in RISC ISAs because with their fixed instruction size there is no way to load a 32-bit constant (or whatever the instruction size is) or larger with one instruction. An `lui` paired with an `or` or some other instruction can load a 32-bit constant. With variable size instructions one can simply have a `pushi` (or other instruction) that uses a 64-bit immediate, there is no need for anything like `lui`.

Inclusion of delayed branches. Delayed branches make sense only in certain pipelined implementations. (Such as those discussed in class so far.) On other implementations delayed branches add to complexity without adding much to performance. (This was mentioned in class several times.) For that reason, delayed branches should have been omitted or their inclusion should have been justified.

Solution to Problem 1a

A feature of stack ISAs and a design goal in this problem is small program size. Small program size is realized by choosing instructions that minimize static instruction count and by coding instructions so they are as small as practical.

Instruction Choice

The choice of instructions was based on those needed for the solution to the last problem in this assignment (for example, add and branch instructions). Other commonly needed instructions were added (for example, xor and store).

Though powerful instructions (for example, those that perform multiple operations such as shift and add) would help reduce the static instruction count they were not added because the problem restricted the ISA to instructions that are no more complex than typical RISC instructions.

Data Types

The ISA uses 64-bit signed and unsigned integers. The memory is byte-addressed and items are in big-endian byte order.

Instruction Coding Overview

To minimize program size the coding was chosen so that as many instructions as possible were only one byte. The Problem 3 solution had several arithmetic and other instructions that used immediate operands. To squeeze them down to one byte the coding was based on a five-bit opcode and a three-bit *extension* field. The extension field holds an immediate or other constant data, or can be used for an extension of the opcode field (as is the function field in MIPS).

Instructions that use the extension field for anything other than an opcode (such as an immediate) are called *Type 1*, the rest are called *Type 2*. Let i_1 denote the number of Type-1 instructions. Clearly $i_1 \leq 32$ and the number of possible Type-2 instructions is $8(32 - i_1)$.

The maximum number of Type 1 instructions is small and so only those instructions which occurred frequently and needed a small (or other) immediate were given Type 1 codings. The other instructions either did not use immediates or had immediates in following bytes.

The size of the extension field was a tradeoff between the number of possible Type 1 instructions and the usefulness of the immediate. (That is, with a 1-bit immediate there could be as many as 128 Type 1 instructions but there would be few cases where the 1-bit immediate would be useful.)

An example of a Type 1 instruction is `add.i`:

```
add.4   # Add 4 to the element at the top of stack.
# Coding of add.4
Field Name:  | opcode  | ext |
Field Value: |       0  | 100 |
Bit Number:   7 6 5 4 3 2 1 0
```

The extension field can hold data other than an immediate. The *subtype* of an instruction specifies what kind of data the immediate field holds. An `add.i` is subtype i. The table below shows the subtypes, the subtypes are explained in detail further below. In the table `EX` refers to the entire extension field (all 3 bits), `EX10` refers to bits 1 and 0 and `EX2` refers to bit 2.

```
1i Use EX as immediate. Whether it's sign extended depends on the instruction.
1v Immediate in following bytes.  EX gives immediate size and padding.
1s Size and padding of data item loaded or stored from memory.
1c Comparison. EX specifies type of comparison.
1t How jump target is determined.
1b Branch condition and size of displacement.
```

```
 1o Use EX for additional opcode bits. (Type 2 instruction.)
```

The assembly language syntax for Type 1 instructions consists of the mnemonic (such as add) followed by a dot and the extension field value, if known, otherwise the subtype name. For example, `add.3` means add the immediate 3 to the TOS while `add.i` refers to a Type 1i add instruction without specifying what the immediate is (as one does when describing the syntax).

Instructions

A complete list of instructions appears below, starting with Type 1 instructions.

Opcode 1	`push.i`	Push immediate on TOS.
Opcode 2	`push.v IMM`	Push immediate on TOS.

Instruction `push.i` pushes value in extension field, i, on the stack. Instruction `push.v` pushes IMM on the stack. IMM is computed using the next 0 to 8 bytes based on the EXT field value as specified in the table below:

```
Type 1v -- Immediate follows first byte of instruction.
EX: Sz: Description
 0:  2: IMM is one byte, no sign extension.
 1:  3: IMM is two bytes, no sign extension.
 2:  5: IMM is four bytes, no sign extension.
 3:  9: IMM is eight bytes.
 4:  2: IMM is one byte, sign extend.
 5:  3: IMM is two bytes, sign extend.
 6:  5: IMM is four bytes, sign extend.
 7:  1: For push, use 0 as value; for others use TOS + 1.

EX is value of extension field.
Sz is total instruction size.
```

```
Execution Examples:
 # 111 222 333 444 555 666
 push.7   # Type 1i
 # 7 111 222 333 444 555 666
 push 0x11  # Type 1v, EX = 0 (one byte immediate, don't sign extend)
 # 0x1 7 111 222 333 444 555 666
 push -10  # Type 1v, EX = 4 (one byte immediate, sign extend)
 # -10 0x1 7 111 222 333 444 555 666
 push 0x123  # Type 1v, EX = 1 (two byte immediate, don't sign extend)
 # 0x123 -10 0x1 7 111 222 333 444 555 666
 push 0x123456789  # Type 1v, EX = 3 (Eight byte immediate. )
 # 0x123456789 0x123 -10 0x1 7 111 222 333 444 555 666
```

```
Coding Examples:

  push.7   # Type 1i
  # Coding of instruction above.
  First Byte
  | opcode  | ext |
  | 00001   | 111 |
    76543     210
  push 0x123  # Type 1v, EX = 1 (two byte immediate, don't sign extend)
  # Coding of instruction above.
  First Byte          Second and Third Byte
  | opcode  | ext |    | IMM            |
  | 00010   | 001 |    | 0x123          |
    00000     000       1111100000000000
    76543     210       5432109876543210
```

Opcode 3 `rollu.i` Roll up by 1, width i.

Pop the TOS and insert it so that it becomes the i'th element.

```
 Example:
 # 111 222 333 444 555 666
 rollu.3
 # 222 333 111 444 555 666

Coding Example:

  rollu.3   # Type 1i
  # Coding of instruction above.
  First Byte
  | opcode  | ext |
  | 00011   | 011 |
    76543     210
```

Opcode 4 `rolld.i` Roll down by 1, width i.

Remove the i'th element and push it on the TOS.

Opcode 5 `index.i` Push a copy of element i (the $(i+1)$'th element).

```
 Example:
 # 111 222 333 444 555 666
 index.0
 # 111 111 222 333 444 555 666
 index.4
 # 444 111 111 222 333 444 555 666

Coding Example:

  index.4   # Type 1i
  # Coding of instruction above.
  First Byte
  | opcode  | ext |
  | 00101   | 100 |
    76543     210
```

Opcode 0 `add.i` Remove the TOS add it to i push the result.

Opcode 6 `add.v IMM` Remove the TOS add it to IMM push the result.

See the Type 1v table above for sizes and padding of IMM.

```
Coding Examples

  add.0   # Type 1i
  # Coding of instruction above.
  First Byte
  | opcode  | ext |
  | 00000   | 000 |
    76543     210

  add.v 0x12345678   # Type 1v
  # Coding of instruction above.
  First Byte           Following four bytes.
  | opcode  | ext |    | IMM                            |
  | 00110   | 011 |    | 0x12345678                     |
    00000     000       33222222222221111111111000000000
    76543     210       10987654321098765432109876543210
```

Opcode 7 `addpower.i` Remove the TOS add 2^i to it and push the result.

Opcode 8 `sub.i` Remove the TOS subtract i from it and push the result.

A sub.v is not included because it would not be used often enough to justify a Type-1 coding.

Opcode 9 `sll.i` Shift left logical.

Opcode 10 `srl.i` Shift right logical.

Opcode 11 `sra.i` Shift right arithmetic.

Remove the TOS, perform the shift by $i+1$ bits and push the result. The assembly language syntax shows the shift amount while the EX field will be coded with the shift amount plus 1. For example, `sll.1` shifts left by one bit and the EX field holds a zero. There is a `shift` instruction for shifts beyond 9 bits.

Opcode 12 `b.b DISP` Branch if TOS non-zero b=1 or if TOS zero (b=0).

A displacement is found in the following 2^{ex10} bytes, the next instruction is the PC plus the displacement.

```
Examples:

  b.1 TARGET    # Branch if TOS non-zero.  TARGET is 20 bytes ahead.
  # Coding of instruction above.
  First Byte           Second Byte
  | opcode  | ext |    | DISP    |
  | 01100   | 100 |    |   10100 |
    00000     000       00000000
    76543     210       76543210

  b.0 TARGET2   # Branch if TOS zero.  TARGET2 is 0x1234 bytes ahead
  # Coding of instruction above.
  First Byte           Second and Third Byte
  | opcode  | ext |    | DISP           |
  | 01100   | 101 |    | 0x1234          |
    00000     000       1111100000000000
    76543     210       5432109876543210
```

Opcode 13 `j.t DISPorTARGET` Jump.

Opcode 14 `jal.t DISPorTARGET` Jump and link.

In both instructions the extension field specifies how to find target address, see the table below. The `jal.t` instructions push a return address on the stack.

```
Type 1t -- Jump Targets
EX: As: Sz: Description
 0: ds: 2 : Displacement target, DISPorTARGET is one byte signed.
 1: ds: 3 : Displacement target, DISPorTARGET is two bytes signed.
 2: ds: 5 : Displacement target, DISPorTARGET is four bytes signed.
 3: ds: 9 : Displacement target, DISPorTARGET is eight bytes signed.
 4: in: 1 : Target is register indirect, address on TOS.
 5: ix: 1 : Target is indexed, sum of top two stack elements.
 6:     1 : Illegal, reserved for future extension.
 7: di: 9 : Direct target, DISPorTARGET is eight bytes unsigned.

EX is value of extension field.
As is assembly language characters for corresponding value.
Sz is the total instruction size, including the first byte.
```

Opcode 15 `cmp.c` Compare.

Opcode 16 `cmpk.c` Compare and keep.

Opcode 17 `cmpz.c` Compare with zero.

Opcode 18 `cmpzk.c` Compare with zero and keep.

Instructions `cmp.c` and `cmpk.c` compare the top two elements using the comparison specified by c (see the table below). Instruction `cmp.c` removes the top two elements `cmpk.c` does not. Instructions `cmpz.c` and `cmpkz.c` are similar except that they compare the TOP element to zero. All instructions push the result of the comparison (zero or one) on the stack.

```
Type 1c -- Conditions. a is TOS, b is 0 or TOS+1
EX: As: Description
 0: eq: a = b
 1: ne: a != b
 2: lt: a<b
 3: le: a<=b
 4: gt: a>b
 5: ge: a>=b
 6: ov: overflow
 7: ca: carry

Note: All instructions are 1 byte.
```

Opcode 19 `loadd.s IMM64` Load direct.

Opcode 20 `stored.s IMM64` Store direct.

Load or store from memory using address IMM64 (the immediate found in the following 8 bytes). The size and padding of the element to load is specified by s. (See the table below.)

Opcode 21	`loado.s OFF8`	Load offset.
Opcode 22	`storeo.s OFF8`	Store offset.

Load or store using the TOS + OFF8 as the address. The size and padding of the element to load is specified by s. (See the table below.)

Opcode 23	`loadr.s`	Load register-indirect.
Opcode 24	`storer.s`	Store register-indirect.

Load or store using the TOS as the address. The size and padding of the element to load is specified by s. (See the table below.)

```
 Type 1s -- Memory access size and padding.
 EX: As:
  0: ub: One byte, unsigned.
  1: uq: Two bytes (quarter word), unsigned.
  2: uh: Four bytes (half word), unsigned.
  3: uw: Eight bytes (word).
  4: sb: One byte, signed. Illegal for stores.
  5: sq: Two bytes, signed. Illegal for stores.
  6: sh: Four bytes, signed. Illegal for stores.
  7:   : Illegal, reserved for future expansion.

Sizes: 9 bytes: loadd.s and stored.s
       2 bytes: loado.s and storeo.s
       1 byte : loadr.s and storer.s



Examples:
  # Stack:  0x1234 111 222
  index.i
  # Stack:  0x1234 0x1234 111 222
  loadr.uw
  # Stack 543210 0x1234 111 222    # 543210 is contents at memory 0x1234.
  rollu.2
  # Stack 0x1234 543210 111 222
  loado.uw 0x10
  # Stack 540000 543210 111 222    # 540000 is contents at memory 0x1244.

Coding Examples:
  loadr.uw
  # Coding of instruction above.
  First Byte
  | opcode  | ext |
  | 10111   | 011 |
    00000     000
    76543     210

  loado.uw 0x10
  # Coding of instruction above.
  First Byte           Second Byte
  | opcode  | ext |    | OFF      |
  | 10011   | 100 |    |   0x10   |
    00000     000       00000000
    76543     210       76543210
```

Type 2 Instructions

There are 24 Type 1 instructions, leaving space for $8(32-24)=64$ Type 2 instructions.

The Type 2 instructions use the extension field as part of the opcode. Some Type 2 instructions have immediates, and some do not.

Opcode 30, Ext 0	`pop`	Pop the stack.

```
Coding Example:
  pop
  # Coding of instruction above.
  First Byte
  | opcode  | ext |
  | 11110   | 000 |
    00000     000
    76543     210
```

Opcode 30, Ext 1	`rolls SHIFT3 SIZE5`	Roll small amount.
Opcode 30, Ext 2	`roll SHIFT8 SIZE8`	Roll.

Rearrange the stack. Instruction `rolls` is two bytes but cannot perform all rolls (on a 32-element stack) whereas `roll` can perform any roll. A size exceeding 32 or a shift exceeding ± 32 is illegal.

Note that the two immediates used by `rolls` fit in one byte.

```
Coding Examples:
  rolls 2 19
  # Coding of instruction above.
  First Byte           Second Byte
  | opcode  | ext |    | SHIFT3 SIZE5 |
  | 11110   | 001 |    | 010    10011     |
    00000     000        000    00000
    76543     210        765    43210
```

Opcode 30, Ext 3	`index DEPTH8`	Push a copy of the stack entry at DEPTH8.
Opcode 30, Ext 4	`sllv`	Shift left logical variable.
Opcode 30, Ext 5	`srlv`	Shift left logical variable.
Opcode 30, Ext 6	`srav`	Shift left logical variable.

The TOS is shift by the amount specified in the low six bits of TOS+1. TOS+1 is removed.

Opcode 30, Ext 7	`shift PAD1 DIR1 AMT6`	Shift. (Combined shift left, right.)

If PAD1 is 1 shift is arithmetic. If DIR1 is 1 shift is left otherwise it is right. AMT6 is the number of bits to shift. Note that the immediates fit in oe byte.

Opcode 29, Ext x	`sub, mul, div`	Arithmetic operations.

Opcode 29, Ext x `and, or, xor` Logical operations.
The indicated operation is performed on TOS and TOS+1.

Opcode 31, Ext x `illegal` Reserved for future second-byte opcode extension.