LSU EE 4720 **Homework 3** Solution **Due: 31 October 2003**

Problem 1: Unlike MIPS, PA-RISC 2.0 has a post-increment load and a load using scaled-index addressing. The code fragments below are from the solution to Problem 2 in the midterm exam the fragments show several MIPS instructions under "Combine" and a new instruction under "Into." For each "Into" instruction show the closest equivalent PA-RISC instructions and show the coding of the PA-RISC instruction. (See the references page for information on PA-RISC 2.0)

(The term *offset* used in the PA-RISC manual is equivalent to the term effective address used in class, and is not to be confused with offset as used in this class. Assume that the `s` field and `cc` fields in the PA-RISC format are zero.)

Show all the fields in the format, including their names and their values.

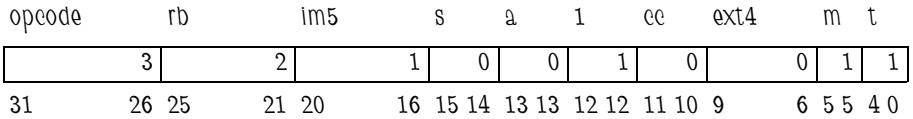
```
Combine:
 lbu $t1, 0($t0)
 addi $t0, $t0, 1
Into:
 lbu.ai $t1, 0($t0)+   # Post increment load.
```

```
; Solution:
  ldb,ma 1(%r2),%r1
  ; %r1 is the equivalent of $t1 above.
  ; %r2 is the equivalent of $t0 above.

  PA-RISC Completer Descriptions:
   m: Modify base register (r2 in example, modify it by adding displacement, 1).
   a: After. (Add the displacement after computing the address.)

  PA-RISC Format 5 Field Descriptions
   opcode: Opcode.
   rb:     Register holding address base. (Address in this case.)
   im5:    Increment amount.  One, to match the MIPS addi instruction.
   s:    * Space register number.  The space registers allow 32-bit programs
           to address more than 4 GiB of memory by holding
           the high 64 bits of a 96-bit address.  Not used in 64-bit code, in
           which case the s field is just used for two more bits of displacement.
   a:      After. If 0, add displ. after load, if 1, add displ. before load.
   l:      Always 1 for format 5 (displacement).
   cc:   * Cache control hint. (0, no hint; 2, spatial locality; 1,3, reserved).
   ext4:   Memory operation.  0 indicates load byte unsigned.
   m:      Modify base register. If 1, write modified address to same register.
   t:      Register in which to write loaded value.

   * You don't need to understand the description of this field.
```



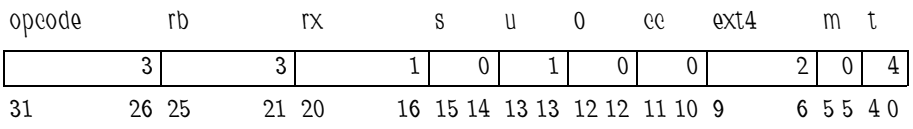
```
Combine:
 sll $t1, $t1, 2
 add $t3, $a1, $t1
 lw $t4, 0($t3)
Into:
 lw.si $t4, ($a1,$t1)  # Scaled index addressing.

; Solution
  ldw.s %r1(%r2), %r4
  ; %r1 is index register (equivalent to $t1 above, before the shift).
  ; %r2 is the base register (equivalent to $a1 above).
  ; %r4 is the destination (equivalent to $t4 above).
  ; Effective address (offset in HP terminology) is: ( %r1 * 4 ) + %r2

 PA-RISC Completer Descriptions:
  s: Scale index.  Multiply the contents of the index register (r1
     here) by the data size, (in this case multiply by 4).


  PA-RISC Format 4 Field Descriptions
   opcode: Opcode.
   rb:     Register holding address base. (Address in this case.)
   rx:     Register holding index.
   s:    * Space register number.  The space registers allow 32-bit programs
           to address more than 4 GiB of memory by holding
           the high 64 bits of a 96-bit address.  Not used in 64-bit code, in
           which case the s field is just used for two more bits of displacement.
   u:      Scale. If 1, shift index by "data size". Shift by 2 for 4-bytes, etc.
   l:      Always 0 for format 4 (indexed addressing).
   cc:   * Cache control hint. (0, no hint; 2, spatial locality; 1,3, reserved).
   ext4:   Memory operation.  2 indicates load word (32 bits) unsigned.
   m:      Modify base register. If 1, write modified address to same register.
   t:      Register in which to write loaded value.

   * You don't need to understand the description of this field.
```



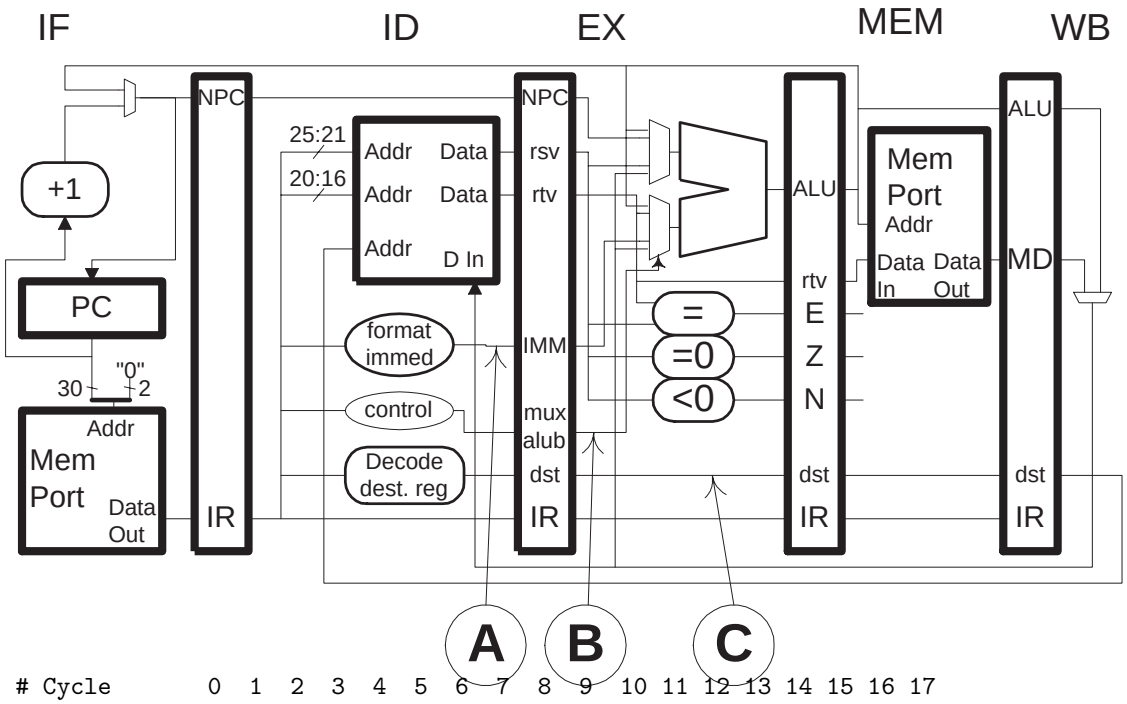
Problem 2: The code fragment below runs on the implementation illustrated below.

(*a*) Show a pipeline execution diagram for the code fragment on the implementation up to the second fetch of the `sub` instruction; assume the branch will be taken.

(*b*) Show the value of the labeled wires (A, B, and C) at each cycle in which a value can be determined.

For maximum pedagogical benefit please pay close attention to the following:

- As always, look for dependencies.
- Pay attention to the RAW hazard between `sub` and `sw` and the RAW hazard between `andi` and `bne`.
- Make sure that `add` is fetched in the right time in the second iteration.
- Base timing **on the implementation diagram**, not on rules inferred from past solutions.




```
LOOP: # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 add r1, r2, r3     IF ID EX ME WB
 sub r3, r1, r4        IF ID EX ME WB
 sw  r3, 0(r5)            IF ID ----> EX ME WB
 andi r6, r3, 0x7            IF ----> ID EX ME WB
 bne r6, $0, LOOP                     IF ID ----> EX ME WB
 addi r2, r2, 0x8                        IF ----> ID EX ME WB
 xor                                              IF IDx (Added to show
 xor                                                 IFx  wrong-path insn.)
LOOP: # Copy of code above.
 add r1, r2, r3                                        IF ID EX ME WB
 sub r3, r1, r4                                           IF ID EX ME WB ...
LOOP: # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 A                  ?  L1 L2 0  0  0  7 -5 -5 -5  8  ?  # ? can't tell;
 B                  ?  ?  t  t  ib ib i  i  ib ib i  i  # t=1, i=2, b=bubble
 C                  ?  ?  1  3  0b 0b 0  6  0b 0b 0  2  # L1=0x0820, L2=0x1822
```

Problem 3: Consider the implementation from the previous problem, repeated below. For the `jr` instruction the ALU sets its output to whatever is at its top input. *Note: This was omitted from the original problem.*

(*a*) There is a subtle reason why the implementation cannot execute a `jr` instruction. What is it? Modify the hardware to correct the problem.

The PC holds bits 31:2 of the address, but the register value sent through the ALU to the PC will be the entire address. If nothing special is done then the jump will be to the address times four. The solution is to have the ALU perform a two bit right shift.

(*b*) There is a reason why it cannot execute a `jalr` instruction. What is it? Modify the hardware to correct the problem.

The **`ex_mem_alu`** pipeline latch is on the path used to put the jump target in the PC and to put the return address in the register file. Therefore, the `jalr` instruction can't do both. A solution would be to add a path from EX to the PC multiplexor for the jump address. Changes shown in **red bold.**