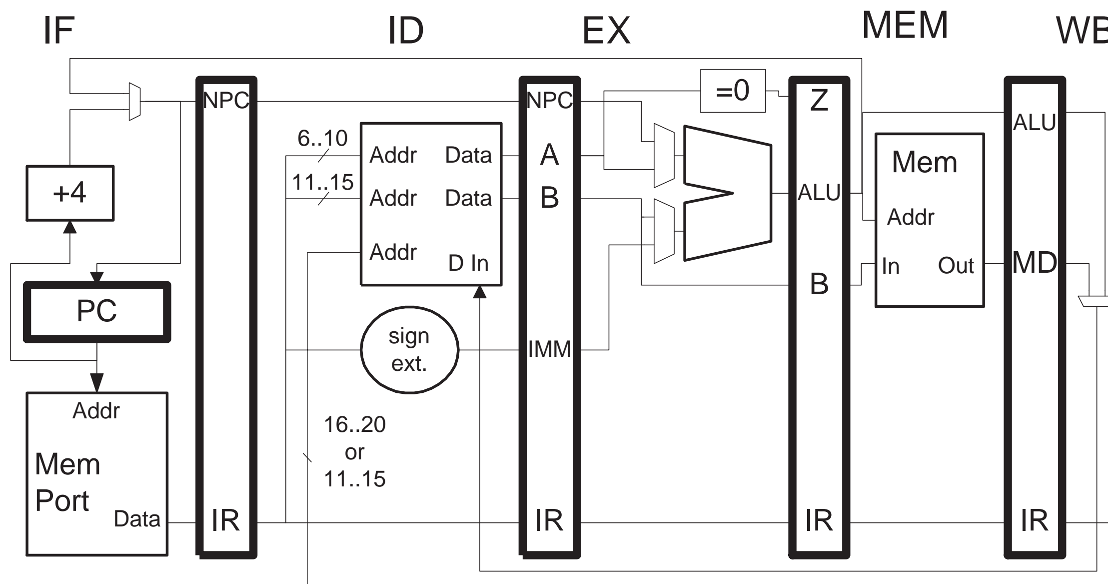
EE 4720 Homework 1 Solution Due: 10 October 2001

Problem 1: In DLX the three instructions below, though they do very different things, are of the same type (format).

```
 bnez r2, SKIP
 lw r1, 1(r2)
 addi r1, r2, #1
SKIP:
```

Because of their similarity their implementations in the diagram below shares a lot of hardware.



(*a*) Show how these DLX instructions are coded.

DLX:

`bnez r2, SKIP`

opcode	rs1→`r2`	rd	simm16
?	2	0	2
0 5	6 10	11 15	16 31

`lw r1, 1(r2)`

opcode	rs1→`r2`	rd→`r1`	simm16
?	2	1	1
0 5	6 10	11 15	16 31

`addi r1, r2, #1`

opcode	rs1→`r2`	rd→`r1`	simm16
1	2	1	1
0 5	6 10	11 15	16 31

(*b*) Find corresponding instructions in the SPARC V9 ISA. (See the SPARC Architecture Manual V9, `http://www.ece.lsu.edu/ee4720/samv9.pdf`) (The DLX branch instruction will have to be replaced by two instructions, one to set the condition code registers.)

```
! In this solution the DLX branch is replaced by a single instruction.
brnz %g1, SKIP
ldsw [%g2+1],%g1
add %g2, 1, %g1

! In this solution the DLX branch is replaced by two instructions.
addcc %g1, 0, %g0
bne SKIP
ldsw [%g2+1],%g1
add %g2, 1, %g1
```

(*c*) Show the coding of the SPARC V9 branch, load, and add immediate instructions (but not the condition code setting instruction).

`brnz g1, SKIP`

op	a	0	rcond	op2	dh	p	rs1	displo
0	0	0	5	3	0	0	2	2
31 30	29 29	28 28	27 25	24 22	21 20	19 19	18 14	13 0

`bne SKIP`

op	a	cond	op2	disp22
0	0	9	2	2
31 30	29 29	28 25	24 22	21 0

`ldsw [g2+1],g1`

op	rd	op3	rs1	i	simm13
3	1	8	2	1	1
31 30	29 25	24 19	18 14	13 13	12 0

`add g2, 1, g1`

op	rd	op3	rs1	i	simm13
2	1	0	2	1	1
31 30	29 25	24 19	18 14	13 13	12 0

(*d*) Do these codings allow the same degree of hardware sharing?

Because the DLX codings are identical an implementation could use the same datapath for computing the immediate add, load effective address, and branch target. The SPARC V9 `add` and `lduw` codings are identical and so hardware can be shared but the placement of the displacement is different for the branch instruction (either one) and so additional hardware would be needed to select the immediate (or displacement) bits corresponding to the instruction.

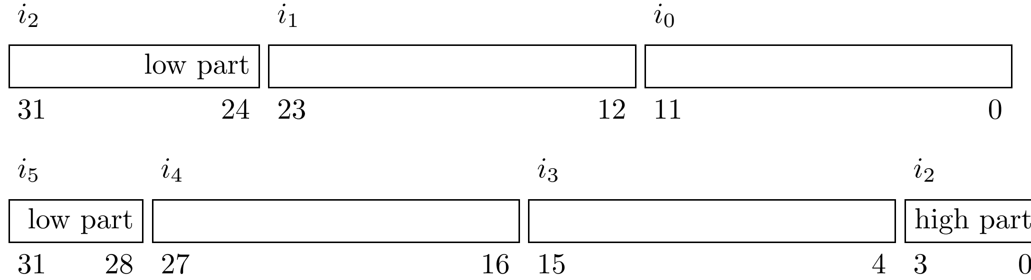
Problem 2: Write a DLX assembly language program that determines the length of the longest run of consecutive elements in an array of words. For example, in array $\{1, 7, 7, 1, 5, 5, 5, 7, 7\}$ the longest run is three: the three 5's (the four 7's are not consecutive). The comments below show how registers are initialized and where to place the longest run length.

```
! r10 Beginning of array (of words).
! r11 Number of elements.
! r1  At finish, should contain length of the longest run.
```

```
! r10 Beginning of array
! r11 Number of elements
! r1  At finish, should contain length of longest run.

! r1 Longest run encountered.
! r2 Size of this run so far.
! r3 Last element.

 add r1, r0, r0
 add r2, r0, r0

 lw r5, 0(r10)
 addi r3, r5, #1

LOOP:
 beqz r11, DONE
 lw r5, 0(r10)
 addi r10, r10, #4
 subi r11, r11, #1
 seq r6, r5, r3
 beqz r6, NEW_RUN
 add r2, r2, #1
 j LOOP

NEW_RUN:
 add r7, r2, r0
 addi r2, r0, #1
 sgt r6, r7, r1
 add r3, r5, r0
 beqz r6, LOOP
 add r1, r7, r0
 j LOOP
```

Problem 3: Small integers can be stored in a packed array to reduce the amount of storage required; the array can be unpacked into an ordinary array when the data is needed. Write a DLX assembly language program to unpack an array containing n b-bit integers stored as follows. The low b bits (bits 0 to $b-1$) of the first word of the packed array contain the first integer, bits b to $2b-1$ contain the next, and so on. When the end of the word is reached integers continue on the second word, etc. Size b is not necessarily a factor of n and so an integer might span two words.

The diagram below shows how the first 6 integers $i_0, i_1, \ldots, i_5$ are stored for $b = 12$ bits and $n \geq 6$.



Write DLX assembly language code to unpack such an array into an array of signed words. The packed array consists of n b-bit signed numbers, with $b \in [1, 32]$. Initial values of registers are given below.

```
! Initial values
! r10: Address of start of packed array.
! r11: Number of elements (n).
! r12: Size of each element, in bits (b).
! r14: Address of start of unpacked array.
```

```
! Initial values
! r10: Address of start of packed array.
! r11: Number of elements (n).
! r12: Size of each element, in bits (b).
! r14: Address of start of unpacked array.

! r1: Current word.
! r2: Mask
! r3: Unpacked item
! r4: Bits remaining in current word.
! r8, r9: Miscellaneous
! r5: 32 - size of each element.

 add r4, r0, r0
 add r8, r12, r0
 addi r5, r0, 32
 sub r5, r5, r12
 addi r2, r0, #1
 sll r2, r2, r12
 subi r2, r2, #1

LOAD_MAYBE:
 bnez r4, LOWPART
 lw r1, 0(r10)
 addi r10, r10, #4
 addi r4, r0, #32

LOWPART:
 and r3, r1, r2
 srl r1, r1, r12
 sub r4, r4, r12

 slt r9, r4, r0
```

```
 bnez r9, SPAN
 j STORE

SPAN:
 lw r1, 0(r10)
 addi r10, r10, #4
 add r8, r12, r4
 sll r9, r1, r8
 or r3, r3, r9
 and r3, r3, r2
 addi r4, r4, #32
 ! Fall through to store

STORE:
 sll r3, r3, r5
 sra r3, r3, r5    ! Sign extend
 sw 0(r14), r3
 addi r14, r14, #4
 subi r11, r11, #1
 bnez r11, LOAD_MAYBE

DONE:
```