EE 4720 **Homework 1** **Due: 10 September 1999**

Problem 1: What are the static and dynamic instruction counts of the two DLX programs below? (DLX is described in Chapter 2 of the text and summarized in the last two pages. Comments, preceded by a `!`, describe what the instructions do.) Be sure to use the value for `r2` specified in the comments. Both programs find the *population* (number of 1's) in the binary representation of the value in `r2`. (For example, the population of $12_{10} = 1100_2$ is 2, $7_{10} = 0111_2$ is 3, and $d06f00d_{16} = 218558477_{10} = 1101000001101111000000001101_2$ is 12.)

```
 ! Program 1.
 ! r2 = 0xd06f00d
 add  r1, r0, r0      ! r1 = 0.  Initialize total.
LOOP:
 andi r3, r2, #1      ! r3 = r2 & 0x1. Put least-significant bit in r3.
 add  r1, r1, r3      ! r1 = r1 + r3.  Add to total.
 srli r2, r2, #1      ! r2 = r2 >> 1.  Shift right logical. Shift off LSB.
 bneq r2, LOOP        ! Branch if r2 not zero. Loop if more.

 ! Program 2.
 ! r2 = 0xd06f00d
 ! r4 = Base of table. Entry i is number of 1's in binary i.
 add  r1, r0, r0      ! r1 = 0. Initialize total.
LOOP:
 andi r3, r2, # 0xff  ! r3 = r2 & 0xff. Put 8 least significant bits in r3.
 add  r5, r4, r3      ! r5 = r4 + r3.   Add to base of population table.
 lbu  r6, 0(r5)       ! r6 = Mem[0+r5]  Load byte unsigned, Load population of r3
 add  r1, r1, r6      ! r1 = r1 + r6.   Add to the total.
 srli r2, r2, #8      ! r2 = r2 >> 8.   Shift right logical. Shift off 8 bits.
 bneq r2, LOOP        ! Loop if r2 not zero.
```

Program 1: static count, 5 instructions. Using data above program iterates 28 times. With four instructions per iteration dynamic count is $1 + 4 \times 28 = 113$ instructions.

Program 2: static count, 7 instructions. Using data above program iterates four times, with six instruction per iteration dynamic count is $1 + 6 \times 4 = 25$ instructions.

Problem 2: Suppose the programs above are run on machines that execute one instruction at a time without overlap (unlike most of the examples shown in class) and with no gaps between. Suppose the CPI for all instructions is 1 cycle and the clock frequency is 625 MHz (period is 1.6 ns). How long would it take each program to run? Suppose the CPI for the `lbu` instruction was 3 cycles. How long would program 2 take?

If all instructions have a CPI of 1, program 1 would take $113\,\text{inst} \times 1\,\text{CPI} \times 1.6\,\text{ns/cycle} = 180.8\,\text{ns}$ and program 2 would take only $40\,\text{ns}$.

With a CPI of three for `lbu`, program 2 would take

$$((1 + 4 \times 5)\,\text{inst} \times 1\,\text{CPI} + 4\,\text{inst} \times 3\,\text{CPI}) \times 1.6\,\text{ns/cycle} = 52.8\,\text{ns},$$

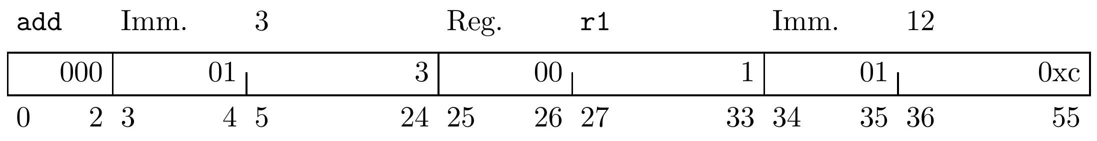
still faster than one. (Program 1 is not affected by the change in CPI for `lbu`.)

Problem 3: What changes would have to be made to program 2 if the `lbu` instruction (load byte unsigned) were changed to `lhu` (load half unsigned)?

The `lbu` instruction loads one byte, the `lhu` instruction loads two bytes. Assume the change was made because the table contains two-byte, rather than one-byte, entries. Then to find the ith index one would look at address $\texttt{r4} + 2 \times i$ rather than $\texttt{r4} + i$. In the program i is the contents of `r3`, so we would have to multiply that by 2. The modified program appears below.

```
 ! Modified Program 2.
 ! r2 = 0xd06f00d
 ! r4 = Base of table. Entry i is number of 1's in binary i.
 add  r1, r0, r0      ! r1 = 0. Initialize total.
LOOP:
 andi r3, r2, # 0xff  ! r3 = r2 & 0xff. Put 8 least significant bits in r3.
 add  r3, r3, r3      ! Multiply r3 by 2. (Using an add for speed.)
 add  r5, r4, r3      ! r5 = r4 + r3.   Add to base of population table.
 lhu  r6, 0(r5)       ! r6 = Mem[0+r5]  Load byte unsigned, Load population of r3
 add  r1, r1, r6      ! r1 = r1 + r6.   Add to the total.
 srli r2, r2, #8      ! r2 = r2 >> 8.   Shift right logical. Shift off 8 bits.
 bneq r2, LOOP        ! Loop if r2 not zero.
```

Problem 4: The Easy ISA as described in class has only five instructions with no *straightforward* way of adding new ones. A non-straightforward way of adding instructions is to take advantage of the fact that the coding does not use all possible combination of bits. In particular, it is possible to specify an immediate as the destination of an arithmetic instruction even though the ISA has no corresponding instruction. For example, consider:



This could be interpreted as instruction `add 3, r1, 12`, however there is no such instruction in the Easy ISA. (If there was, what would it do?)

Explain how this "hole" can be used to code additional instructions. Use this coding to add `and`, `or`, `sll` (shift left logical), and `srl` (shift right logical) instructions. The new instructions should use the same addressing modes as the existing arithmetic instructions.

If the addressing mode for the destination is immediate, interpret the immediate value as an extended opcode and interpret the next three operand fields as destination, source 1, and source 2. Codings for the logical instructions: `and`, 0; `or`, 1; `sll`, 2; and `slr`, 3.

Problem 5: Recall that an issue (it's not okay to say problem anymore) with the Easy ISA is that there is no CTI (control-transfer instruction: branch, jump, call, return, etc.) that will branch to an address held in a register. Only self-modifying code can do that. Write such code. The code should branch to an address held in register `r100`. The solution may use the instructions added above. Addresses in Easy ISA do not have to be aligned. Assume the most significant bit of the address is always zero. *Hint: This assumption and the lack of alignment restrictions makes things alot easier.*

The branch instruction contains its target address in bits 15 through 78. Those bits will have to be overwritten with the target address held in `r100`. Call the address of the branch instruction `BLINE`. All Easy ISA instructions write 64-bit words starting at any address. Suppose `r0` held a zero. Instruction `add [BLINE+r0], r10, 0` would write the contents of `r10` to the first 64 bits of the branch instruction. If `r1` holds a 1 then `add [BLINE+r1], r10, 0` would write the contents of `r10` to bits 8 through 71 of the branch instruction. If `r2` held a 2 then `add [BLINE+r2], r10, 0` would modify bits 16 through 79 which is almost what we need. Since there is no way to exactly write bits 15 through 78, the target address will have to be prepared. In this case preparation merely consists of shifting it one position to the left. Because the MSB of addresses are always zero there is no need to modify bit 15 and nothing need be done with the MSB bit of r100. The solution appears below:

```
 add r3, 0, 2   ! Put constant 2 in r3.
! Shift the address by 1, store in branch, starting at byte 2.
 sll [r3+BLINE], r100, 1
 add r1, 0, 1   ! Set branch condition.
BLINE:
 b r1, r101, DONTCARE   ! DONTCARE is changed by the time it executes.
```