LSU EE 4720 **Homework 4** Solution **Due: 22 March 2004**

Problem 1: Suppose code like the memory copy program below (from Homework 3) appears frequently enough in the execution of programs so that new instructions should be added to the ISA to allow improved execution. (It does and they have been.)

Following the points below devise new instruction(s) that can be used to write a new memory copy loop that would execute more efficiently than is possible with existing MIPS-I instructions. A goal is to copy at the rate of two bytes per cycle. See the subparts after the bulleted points below.

- The instructions must use the existing MIPS formats.
- An instruction can do more than one thing (as long as it follows the points below). For example, an instruction that does more than one thing is a post-increment load. To reach the two bytes / cycle limit one might need to combine a branch with something.
- The instructions cannot use implicit registers. (A register is implicit if it does not appear in the encoded instruction. For example, register 31 is implicit in the `jal` instruction.)
- To achieve two bytes per cycle the instructions might need to do something unusual with operands. Please ask if you're not sure if something is too unusual.
- As with all other ordinary instructions, the new instructions must advance one stage per cycle (unless stalled, if so they would sit idle).
- The modified pipeline must still use the same memory port and no new memory ports can be added.
- Modifications such as bypass paths can be added to speed the instructions.

(*a*) Show an example of each new instruction and show how it is coded.

(*b*) Show how the instructions would be implemented on the pipeline.

(*c*) Write a memory copy program using the new instructions.

(*d*) Show a pipeline execution diagram for the memory copy code.

(*e*) A two bytes per cycle solution would require doing something interesting for the branch. Explain what that is and show a pipeline execution diagram for the memory copy loop finishing a copy (where the interesting stuff would be done).

Solution starts on next page (not counting this sentence).

The solution adds two instructions, an indexed-looping (IL) load, and an indexed-incrementing (II) store:

```
 ($s0)  lw,il  $t0, ($a2-$s0)
 ($s0-) sw,ii  ($a3-$s0), $t0
```

The IL load, a Format R instruction, has two source operands, a base (`rs`, in the example `a2`) and an index (`rd`, in the example `s0`), and a destination (`rt`, in the example `t0`).

If the index is zero the instruction does nothing (it acts like a `nop`). If the index is non-zero then it loads the value at address base - index (in the example `a2-s0`) into register `rt`. It also does a delayed control transfer to its own address (it branches to itself). In the example, the `lw,il` will jump to itself (with `sw,il` executing in the delay slot). Note that the `rd` register works something like a predicate, which is why it is written using the syntax of predicated instructions (the `(s0)` at the start of the instruction).

The indexed-incrementing store is also a Format R instruction, it has three source operands, a base (`rs`), an index (`rd`), and a store value (`rt`). If the index is zero the instruction does nothing. Otherwise, it stores the `rt` value at address `rsv - rdv` and it writes register `rd` with `rdv - 4`. (Unlike most other MIPS instructions, a single register field (`rd`) is being used to specify both a source and destination.) Note that the `lw,il` and `sw,ii` instructions compute their effective addresses in the same way but that the `sw,ii` decrements the index while the `lw,il` does not change the index.

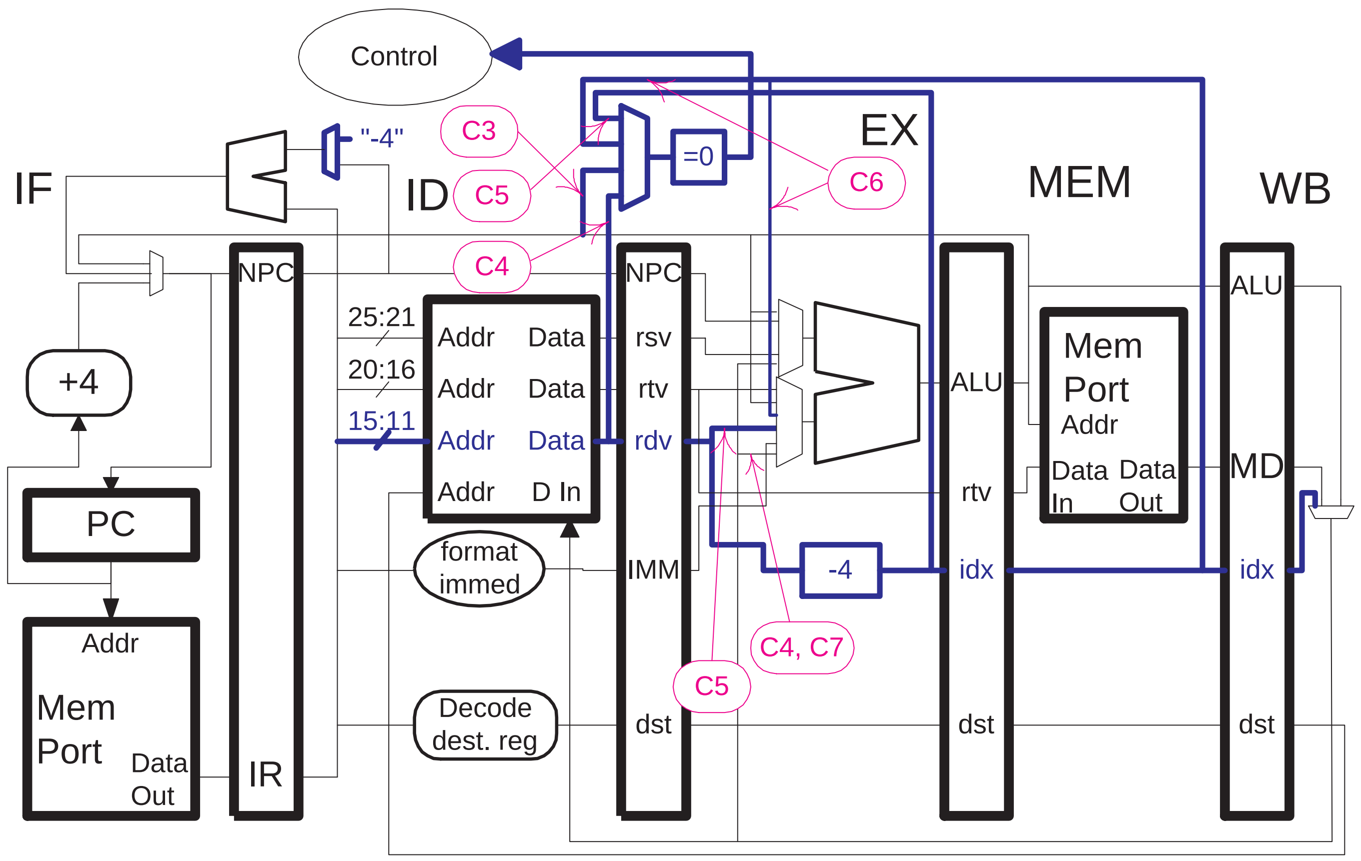
The program below uses the new instructions to copy a region of memory. The program starts with the same register values (`a0`, `a1`, and `a2`) as the original program and does the same thing. Unlike the original program it uses two instructions before the loop. The first computes the size of the region to copy. The second computes the end of the region to copy data to. (The end of the region to copy data from is already provided, in `a2`.) The loop label (`LOOP`) is shown for illustrative purposes, but the assembler ignores it because the `lw,il` always branches to itself.)

```
 # $a0 Start address of region to copy.
 # $a1 Address of memory to copy to.
 # $a2 Address at end of region to copy. (Don't copy $a2, do copy $a2-4.)

 sub $s0, $a2, $a2              # Size of region to copy.
 add $a3, $a1, $s0              # Address at end of region to copy to.
LOOP:
 ($s0)  lw,il  $t0, ($a2-$s0)  # Load word and branch if $s0 not zero.
 ($s0-) sw,ii  ($a3-$s0), $t0  # Store and decrement $s0.

 # Same program, with pipeline execution diagram.

 # Cycle                    0  1  2  3  4  5  6  7  8  9
 sub $s0, $a2, $a2          IF ID EX ME WB
 add $a3, $a1, $s0             IF ID EX ME WB
LOOP:
 ($s0)  lw,il $t0, ($a2-$s0)      IF ID EX ME WB          (1st iteration)
                                        IF ID EX ME WB    (2nd iteration)
 ($s0-) sw,ii ($a3-$s0), $t0         IF ID EX ME WB       (1st iteration)
                                           IF ID EX ME WB (2nd iteration)
 # Cycle                    0  1  2  3  4  5  6  7  8  9
```

The diagram below shows how the new instructions might be implemented, changes are shown in blue. A third read port is added to the register file so that the store instruction can read the index, base, and store value simultaneously. A comparison unit is added to the ID stage to check for the end of loop condition (index zero); note that several bypass connections are needed. A decrementer (-4) is added to the EX stage (used by `sw,ii`) and pipeline latch registers are added to pass the new value of the index down the pipeline.

The cycle numbers, in purple show when the labeled lines will be used for the pipeline execution diagram above.

The hardware for predication is not shown. That hardware would replace the `dst` value with a zero and change the memory operation to a nop. (The memory operation input is also not shown.)


Control
IF
ID
EX
MEM
WB
"-4"
=0
C3
C5
C4
C6
NPC
IR
+4
PC
Addr
Mem Port
Data Out
25:21
20:16
15:11
Addr Data
Addr Data
Addr Data
Addr D In
format immed
Decode dest. reg
NPC
rsv
rtv
rdv
IMM
dst
-4
C4, C7
C5
ALU
rtv
idx
dst
Mem Port
Addr
Data In
Data Out
ALU
MD
idx
dst