EE 4720 Homework 2 Solution Due: 5 November 2001

Problem 1: Answer the following questions about the MIPS Technologies 4Km processor core. The processor is documented in
`http://www.mips.com/declassified/Declassified_2000/MD00016-2B-4K-SUM-01.15.pdf`.

(*a*) For each stage in the statically scheduled DLX implementation show where the same work is done in the 4Km pipeline. Note that work done in one DLX stage might be performed in more than one 4Km pipeline stage.

IF: I

ID: E

EX: Part of the ALU operation done in E, part in M. Address calculations for load and stores done in E.

MEM: Memory load and store done in M stage. Alignment done in A stage.

WB: W.

(*b*) The 4Km documentation uses the term *stall* differently than used in class. How do their usages differ? What term does the documentation use that is close to stall as used in class? (See section 2.8.1)

By stall the 4Km documentation means take more than one cycle to complete a computation, as do the floating-point units in DLX. Unlike the use in class, it does not mean that instructions following (more recently fetched than) the stalled instruction are stopped. The documentation uses the term *slip* for what is meant by stall in class.

(*c*) A MIPS implementation needs to do all of the following:

(1) arithmetic and logical operations for ordinary instructions
(2) compute the target of a branch
(3) compute the effective address of a load or store

In the first pipelined DLX implementation all of these were performed by the ALU. MIPS has a branch instruction in which a branch is taken if two registers are equal (`beq`) or not equal (`bne`). So it must also

(4) determine if two values are equal

How many of these are shared? If they are not shared, why not? (The documentation does not state exactly what hardware is present, answer the question by looking at how instructions execute.)

The ALU, effective address computation, and part of a branch target computation (I-AC2) may be shared. All of these are done in the second half of E (the ALU is also used in the first half of M). An instruction needs to do at most one of these things. (e.g., load or store instructions, which compute effective addresses, do not need to compute branch targets or need to use the ALU for other arithmetic or logical operations.) Therefore these [(1), (2), and (3)] can be shared.

According to 2.1.2 an instruction address is determined in E, and so the condition must be evaluated in E. Register values are not available until the second half of E so the register comparison to determine the branch condition must also be evaluated in the second half of E, the same time as branch target address computation (assuming that's what I-AC2 does). Therefore separate hardware is needed for the branch condition.

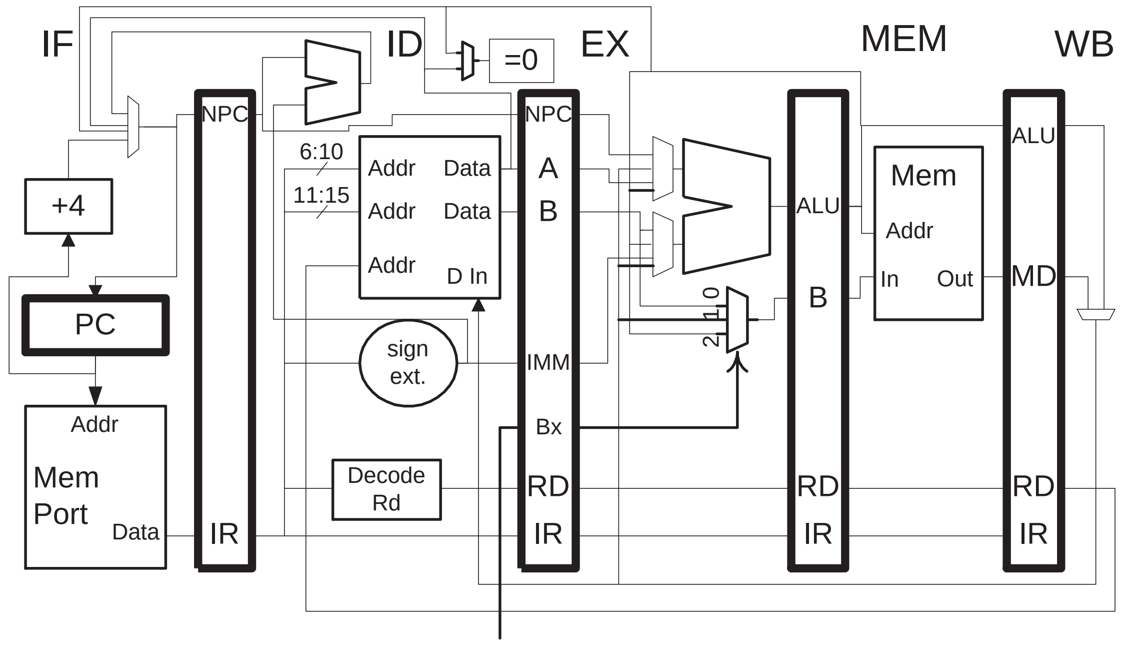
Problem 2: The program below runs on the DLX implementation shown below. The hardware makes no special provisions for the tricky technique used. The coding for a `nop` (actually `add r0, r0, r0`) is all zeros.

Why isn't this an infinite loop? (For those who know why it matters, assume there is no cache.)

The `sw` instruction replaces the `j` instruction with a `nop`.

Why will the code run for at least two iterations?

Because in the first iteration the `sw` instruction reaches MEM after the `j` is already fetched. If a write to the MEM-stage memory port is seen by the IF-stage memory port then the loop will perform only two iterations.



```
LOOP:
 lw r1, 0(r2)
 addi r2, r2, #4
 add r3, r3, r1
 sw 0x100(r0), r0
LINE:  LINE = 0x100
 j LOOP
```

Problem 3: Show a pipeline execution diagram for the code below running on a 4-way statically scheduled superscalar processor. All needed bypass paths are available, including one for the branch condition. Determine the CPI for a large number of iterations.

```
 and r2, r2, r8
LOOP:  ! LOOP = 0x1008
 lw r1, 0(r2)
 add r3, r3, r1
 addi r2, r2, #4
 sub r4, r2, r5
 bneq r4, LOOP
```

Based on the PED below the CPI is $\frac{7}{5} = 1.4$. The pipeline execution diagram is for the second (or later) iteration.

```
 and r2, r2, r8
LOOP:  ! LOOP = 0x1008
 ! Cycle          0  1  2  3  4  5  6  7
 lw r1, 0(r2)     IF ID EX ME WB       IF
 add r3, r3, r1   IF ID ----> EX ME WB IF
 addi r2, r2, #4     IF ----> ID EX ME WB
 sub r4, r2, r5      IF ----> ID -> EX ME
 bneq r4, LOOP       IF ----> ID ----> EX
```

Problem 4: The code from the problem above can be improved (stalls can be removed) to a small extent by scheduling, but that would still leave some stalls. This might see like a good candidate for loop unrolling.

(*a*) Show why it would take alot of unrolling to eliminate all stalls. (You don't have to show the unrolled code, since it would be long.)

Because of the 1-cycle load latency the consuming add instruction would have to be placed seven instructions away. Two of those can be an **addi** and **sub**, the rest would be **lw**, so the loop would be unrolled six times. This is shown below. The code has been slightly re-structured To facilitate unrolling positions of the **sub** and **addi** have been switched, with a compensating instruction added before the loop. To avoid added dependencies six running sums are computed, at the end of the loop these are added together.

```
 and r2, r2, r8
 subi r5, r5, #24  ! Compensate for switching position of sub and addi below.
 nop
LOOP:  ! LOOP = 0x1010
 lw r1,  0(r2)    IF ID EX ME WB
 lw r11, 4(r2)    IF ID EX ME WB
 lw r12, 8(r2)    IF ID EX ME WB
 lw r13, 12(r2)   IF ID EX ME WB
 lw r14, 16(r2)      IF ID EX ME WB
 lw r15, 20(r2)      IF ID EX ME WB
 sub r4, r2, r5      IF ID EX ME WB
 addi r2, r2, #24    IF ID EX ME WB
 add r3, r3, r1         IF ID EX ME WB
 add r21, r21, r11      IF ID EX ME WB
 add r22, r22, r12      IF ID EX ME WB
 add r23, r23, r13      IF ID EX ME WB
 add r24, r24, r14         IF ID EX ME WB
 add r25, r25, r15         IF ID EX ME WB
 bneq r4, LOOP             IF ID EX ME WB
```

```
! Note: Could add differently to avoid stalls.
 add r3, r3, r21          IF IDx
 add r3, r3, r22             IFx
 add r3, r3, r23             IFx
 add r3, r3, r24             IFx
 add r3, r3, r25             IFx
```

(*b*) Use software pipelining and scheduling to remove the stalls. (Hint: to software pipeline switch the `lw` and `add` instructions, and make any other necessary changes.) What is the CPI for a large number of iterations of the modified code?

The loop below runs with a CPI of $\frac{3}{5} = 0.6$. The `add` and `lw` were switched and prolog and epilog code, instructions before and after the loop to compensate, was added. Software pipelining was also used for the branch condition: the `sub` and `addi` were reversed.

```
 and r2, r2, r8
 add r1, r0, r0
 subi r5, r5, #4
LOOP:  ! LOOP = 0x1010
 ! Cycle          0  1  2  3  4
 add r3, r3, r1   IF ID EX ME WB
                           IF ID
 lw r1, 0(r2)     IF ID EX ME WB
                           IF ID
 sub r4, r2, r5   IF ID EX ME WB
                           IF ID
 addi r2, r2, #4  IF ID EX ME WB
                           IF ID
 bneq r4, LOOP       IF ID EX ME
 add r3, r3, r1      IF IDx
```

(*c*) Would loop unrolling provide further gains?

It always does. As always, unrolling would reduce the proportion of loop index instructions (those computing the address of the load and the branch condition). Unrolling might place the branch in the last position in a group, reducing fetch waste. Because there are fewer iterations, it will reduce the number of instructions squashed due to the taken branch.