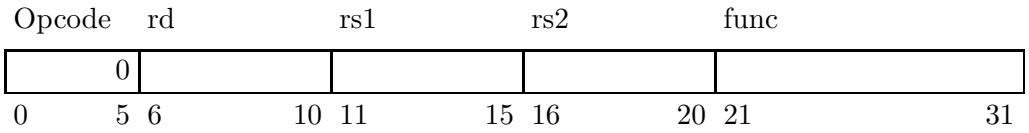
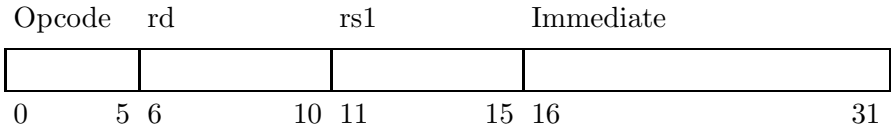
**EE 4720** **Homework 2** Solution **Due: 4 October 1999**

**Problem 1:** Suppose the coding of DLX instructions were changed so the destination appeared before the source operands, as shown in the codings below:

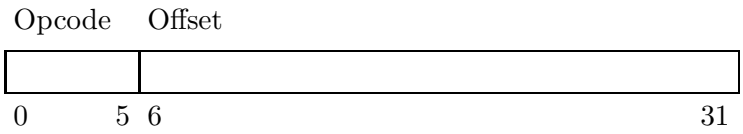
New Type R:




New Type I:



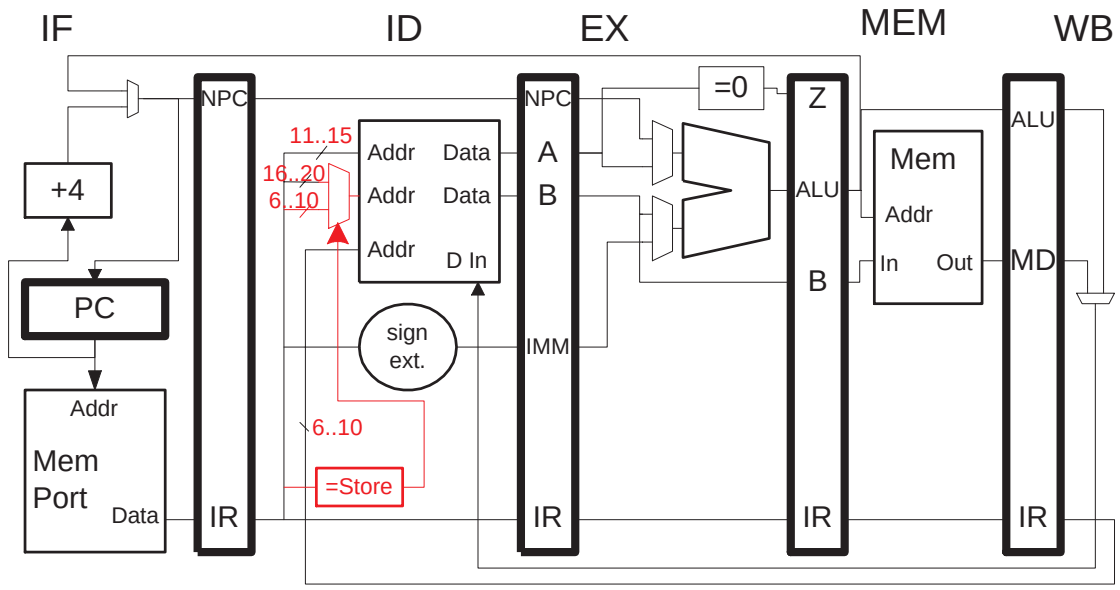

Type J: (no change)




Show the changes needed to the pipeline below to implement this new ISA. The changes should only effect the ID and WB stages. If there are differences in the control inputs to multiplexors or other units, explain what those differences are.

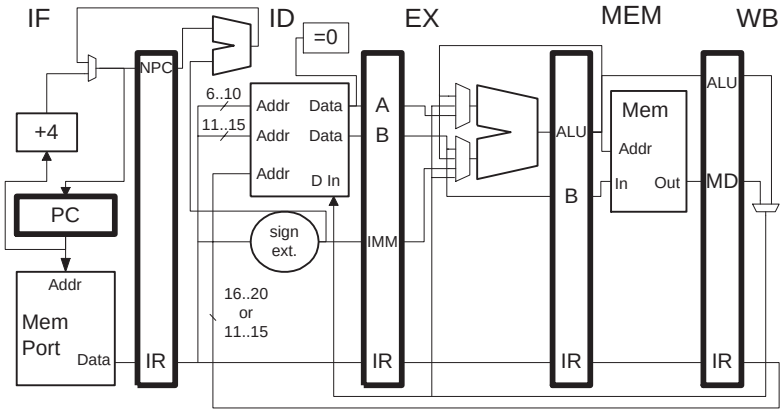
**Make sure your design executes store instructions correctly.**

Changes shown in red.



**Problem 2:** The program below executes on the DLX implementation shown below. The implementation uses forwarding (bypassing) to avoid some data hazards and stalls to avoid others. All forwarding paths are shown. (If a needed forwarding path is not there, sorry, you'll have to stall.) A value can be read from the register file in the same cycle it is written. The destination field in the `beqz` is zero. Instructions are nulled (squashed) in this problem by replacing them with `slt r0,r0,r0`. All instructions stall in the ID stage.

```
 ! Initially, r1=0x101, r2=0x202, r3=0x303
 ! MEM[0x103] = 0xfe
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, r0, r0
START: ! START = 0x50
 lb   r1, 2(r1)
 addi r1, r1, #3
 or   r1, r1, r2
 beqz r2, SKIP !(taken)
 add  r3, r1, r2
 sub  r0, r0, r0
 sub  r0, r0, r0
SKIP:
 xor  r3, r1, r3
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, r0, r0
```




The table below shows the contents of pipeline registers and changes to architecturally visible registers `r1-r31` over time. Cycle zero is the time that `lb` is in instruction fetch. The first two columns are completed; fill in the rest of the table. Use a "?" for the value of the "immediate field" of a type R instruction and for the output of the memory when no memory read is performed. Show pipeline register values even if they're not used. Assume that the ALU performs the branch target computation even though it was already computed in ID. The row labeled "Reg. Chng." shows a new register value that is available at the *beginning* of the cycle. If no register value is written leave the entry blank.

*Hints: See Spring 1999 HW 3 for a similar problem. One feature of the solution would not be present if* `lb` *were replaced by a* `addi`*. Another feature may not be present if* `lb` *were replaced by* `lw`*.*

Completed table appears below.

| Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| PC | 0x50 | 0x54 | 0x58 | 0x58 | 0x5c | 0x60 | 0x6c | 0x70 | 0x74 | 0x78 | 0x7c |
| IF/ID.IR | sub | lb | addi | addi | or | beqz | add | xor | sub | sub | sub |
| Reg. Chng. | r0 ←0 | r0 ←0 | r0 ←0 | r0 ←0 | r0 ←0 | r1 ←-2 | r0 ←0 | r1 ←1 | r1 ←203 | r0 ←0 | r0 ←0 |
| ID/EX.IR | sub | sub | lb | slt | addi | or | beqz | slt | xor | sub | sub |
| ID/EX.A | 0 | 0 | 0x101 | 0 | 0x101 | -2 | 0x202 | 0 | 0x203 | 0 | 0 |
| ID/EX.B | 0 | 0 | 0x101 | 0 | 0x101 | 0x202 | ? | 0 | 0x303 | 0 | 0 |
| ID/EX.IMM | ? | ? | 2 | ? | 3 | ? | 3 | ? | ? | ? | ? |
| EX/MEM.IR | sub | sub | sub | lb | slt | addi | or | beqz | slt | xor | sub |
| EX/MEM.ALU | 0 | 0 | 0 | 0x103 | 0 | 1 | 0x203 | 0x6c/4 | 0 | 0x100 | 0 |
| EX/MEM.B | 0 | 0 | 0 | 0x101 | 0 | 0x101 | 0x202 | 0 | 0 | 0 | 0 |
| MEM/WB.IR | sub | sub | sub | sub | lb | slt | addi | or | beqz | slt | xor |
| MEM/WB.ALU | 0 | 0 | 0 | 0 | 0x103 | 0 | 1 | 0x203 | 0x6c/4 | 0 | 0x100 |
| MEM/WB.MD | ? | ? | 0 | 0 | -2 | ? | ? | ? | ? | ? | ? |

To help solve the problem, find a pipeline execution diagram for the code (shown below). Cycle numbers in diagram and table match.

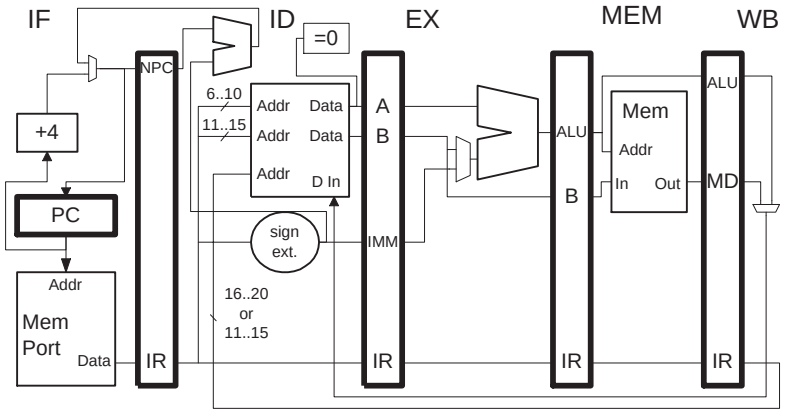
```
 ! Initially, r1=0x101, r2=0x202, r3=0x303
 ! MEM[0x103] = 0xfe
 sub  r0, r0, r0
 sub  r0, r0, r0
! Cycle              0   1   2   3   4   5   6   7   8   9   10
START: ! START = 0x50
 lb   r1, 2(r1)      IF  ID  EX  MEM WB
 addi r1, r1, #3         IF  ID  --> EX  MEM WB
 or   r1, r1, r2             IF  --> ID  EX  MEM WB
 beqz r2, SKIP !(taken)              IF  ID  EX  MEM WB
 add  r3, r1, r2                         IFx
 sub  r0, r0, r0
 sub  r0, r0, r0
SKIP:
 xor  r3, r1, r3                             IF  ID  EX  MEM WB
 sub  r0, r0, r0                                 IF  ID  EX  MEM
 sub  r0, r0, r0                                     IF  ID  EX
 sub  r0, r0, r0                                         IF  ID
```

**Problem 3:** Consider the program:

```
LOOP:
 lw   r1, 0(r2)
 add  r3, r1, r3
 addi r2, r2, #4
 bneq r1, LOOP
 or   r4, r5, r6
```

For each implementation below provide a pipeline execution diagram showing execution up to the third fetch of `lw` and determine the CPI for a large number of iterations.

Not Bypassed:

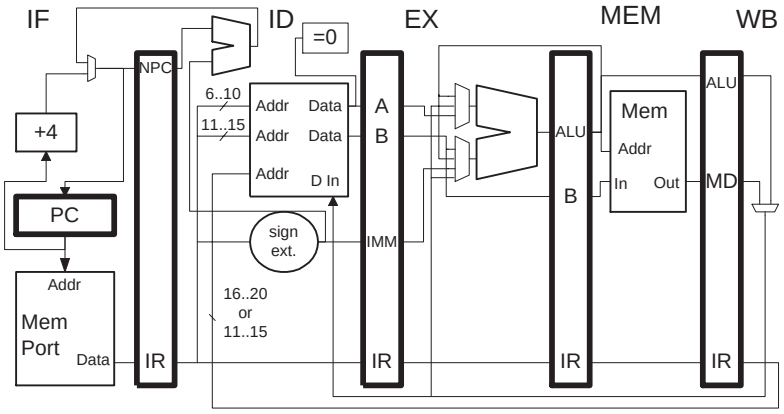



Solution:

```
! Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
LOOP:
 lw   r1, 0(r2)   IF  ID  EX  MEM WB          IF  ID  EX  MEM WB          IF
 add  r3, r1, r3      IF  ID  -----> EX  MEM WB  IF  ID  -----> EX  MEM WB
 addi r2, r2, #4          IF  -----> ID  EX  MEM WB  IF  -----> ID  EX  MEM WB
 bneq r1, LOOP                       IF  ID  EX  MEM WB          IF  ID  EX  MEM WB
 or   r4, r5, r6                         IFx                         IFx
```

Each iteration takes the same amount of time, 7 cycles, and contains 4 instructions, for a CPI of $\frac{7}{4}$ CPI $= 1.75$ CPI.

Bypassed:




Solution:

```
! Cycle          0   1   2   3   4   5   6   7   8   9   10  11  12
LOOP:
 lw   r1, 0(r2)  IF  ID  EX  MEM WB      IF  ID  EX  MEM WB      IF
 add  r3, r1, r3     IF  ID  --> EX  MEM WB  IF  ID  --> EX  MEM WB
 addi r2, r2, #4         IF  --> ID  EX  MEM WB  IF  --> ID  EX  MEM
 bneq r1, LOOP               IF  ID  EX  MEM WB      IF  ID  EX
 or   r4, r5, r6                 IFx                     IFx
```

Each iteration takes the same amount of time, 6 cycles, and contains 4 instructions, for a CPI of $\frac{6}{4}$ CPI $= 1.5$ CPI.

**Problem 4:** Schedule (rearrange) the instructions in the program used in the previous problem to improve execution speed. (Do not change what the program does!). Show pipeline execution diagrams and determine CPI for the two implementations.

Solution:

```
! Not bypassed.
! Cycle          0   1   2   3   4   5   6   7   8   9   10  11  12
LOOP:
 lw   r1, 0(r2)  IF  ID  EX  MEM WB      IF  ID  EX  MEM WB      IF
 addi r2, r2, #4     IF  ID  EX  MEM WB      IF  ID  EX  MEM WB
 add  r3, r1, r3         IF  ID  ->  EX  MEM WB  IF  ID  ->  EX  MEM
 bneq r1, LOOP               IF  ->  ID  EX  MEM WB  IF  ->  ID  EX
 or   r4, r5, r6                 IFx                     IFx
```

Each iteration takes the same amount of time, 6 cycles, and contains 4 instructions, for a CPI of $\frac{6}{4}$ CPI $= 1.5$ CPI.

Solution:

```
! Bypassed Implementation
! Cycle          0   1   2   3   4   5   6   7   8   9   10  11  12
LOOP:
 lw   r1, 0(r2)  IF  ID  EX  MEM WB  IF  ID  EX  MEM WB  IF
 addi r2, r2, #4     IF  ID  EX  MEM WB  IF  ID  EX  MEM WB
 add  r3, r1, r3         IF  ID  EX  MEM WB  IF  ID  EX  MEM
 bneq r1, LOOP               IF  ID  EX  MEM WB  IF  ID  EX
 or   r4, r5, r6                 IFx                 IFx
```

Each iteration takes the same amount of time, 5 cycles, and contains 4 instructions, for a CPI of $\frac{5}{4}$ CPI $=$ $1.25$ CPI.

**Problem 5:** Show the changes needed to implement the predicated instructions presented in class. (Set 4, page 25, as of this writing.) Describe the instruction format and show any datapath and control changes to the implementation below.

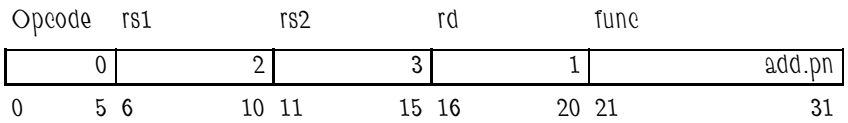
The solution described below adds predicated type-R instructions.

First, an instruction coding needs to be found. The coding should fit naturally into the DLX ISA such that implementations would be changed as little as possible. Since this is an addition to DLX, existing DLX instructions must not be changed.

Predicated versions of type-R instructions will be added. The predicated instructions have new opcodes, the new opcodes will not be listed. (As with other type-R instructions, the opcode is in the Func field.)

Unless the format is changed, there is no room to specify the predicate register. Rather than changing the format, the interpretation of the fields will be changed. The destination filed (⟨**rd**⟩) will specify the predicate and ⟨**rs1**⟩ will specify the first source operand (as usual) and the destination (they will always be the same). For example, instruction `(r1) add r2, r2, r3` is coded:

Type R:




where opcode `add.pn` indicates a predicated add which writes its result of the predicate is non-zero. (If the Func field contained an ordinary add the instruction would be `add r1, r2, r3`.)

Here are some not-so-good alternative codings: Add a third source operand field, increasing the instruction size to 40 bits (maybe use the 3 left over bits for more opcode space). If all instructions are 40 bits, then old code won't work and so this is really a new ISA, not an extension of an existing one. If only predicated instructions are 40 bits, then implementation will be a challenge. First (this will be covered later in the semester) it's alot harder to build a memory system that returns any five consecutive bytes. It's much easier to fetch a power-of-two bytes at an aligned address. Another problem is that before the PC is incremented one has to find the instruction size, in the implementations considered size is determined in the cycle after its needed. If the PC were incremented in the beginning of the fetch cycle we could determine whether the previous instruction (in ID) was predicated, but the IF critical path length would be long in that case.

Now that the coding is determined, the pipeline must be modified to implement it. Predicated instructions need three register values, that can't be avoided so a third read port must be added to the register file (see the illustration below). (Some real ISAs have special predicate registers, so additional general purpose register file ports are not needed.)

Predicated instructions have the destination register in a different place (bits 6 to 10) than other type-R instructions (16 to 20). The decode logic must recognize predicated instructions and place the correct destination register in the `ID/EX.RD` pipeline latch. (See illustration.)

An instruction is called predicated because its result isn't written back if the predicate is false. This will be implemented by replacing the destination register with a 0 in the EX stage. An `=0` checks the predicate to see if it's zero. The predicate may come from the register file or be bypassed from MEM or WB. In ordinary predicated instructions the predicate is false if the predicate register is zero. In inverted predicate instructions (`(!r1) add r2, r2, r3`) the predicate is false if the predicate register is non-zero. An exclusive or gate is used to invert the output of `=0` for inverted instructions. (The output of `=Pred 0` is true if an inverted predicated instruction is present.)

If the predicate were tested in ID then it would not be possible to use the result of an immediately preceding instruction.

Changes are shown in red: