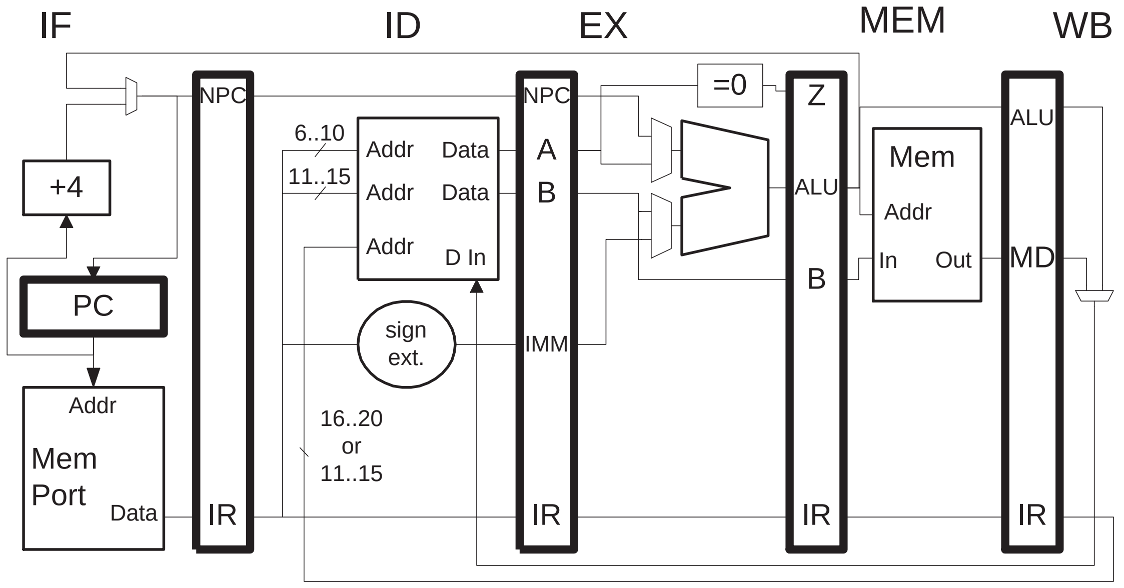
EE 4720 **Homework 3** **Due: 8 March 1999**

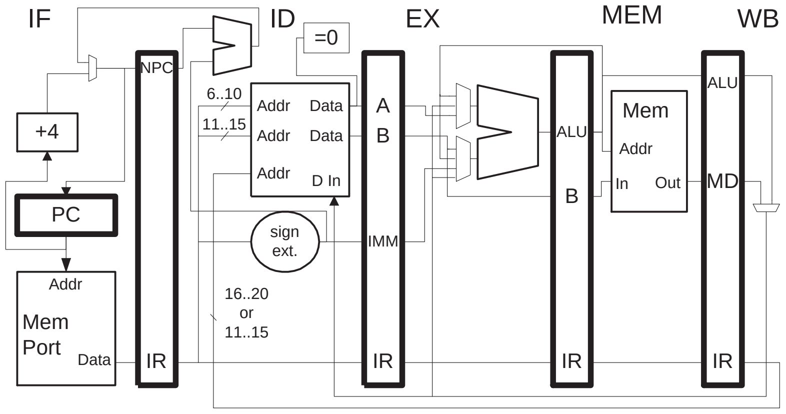
In all problems below assume there are no cache misses and that all register values are available at the beginning of execution.

Problem 1: The pipeline shown below cannot execute the `jal` or `jalr` instructions. Identify and fix the problem. *(Hint: Think about a difference between* `jal` *and* `beqz` *besides the fact that* `jal` *is unconditional.)*



Problem 2: The program below executes on the DLX implementation shown below. The comments show the results of some instructions. The implementation uses forwarding (bypassing) to avoid some data hazards and stalls to avoid others. The forwarding paths are shown. A value can be read from the register file in the same cycle it is written. The destination field in the `bneq` is zero. Instructions are nulled (squashed) in this problem by replacing them with `or r0,r0,r0`.

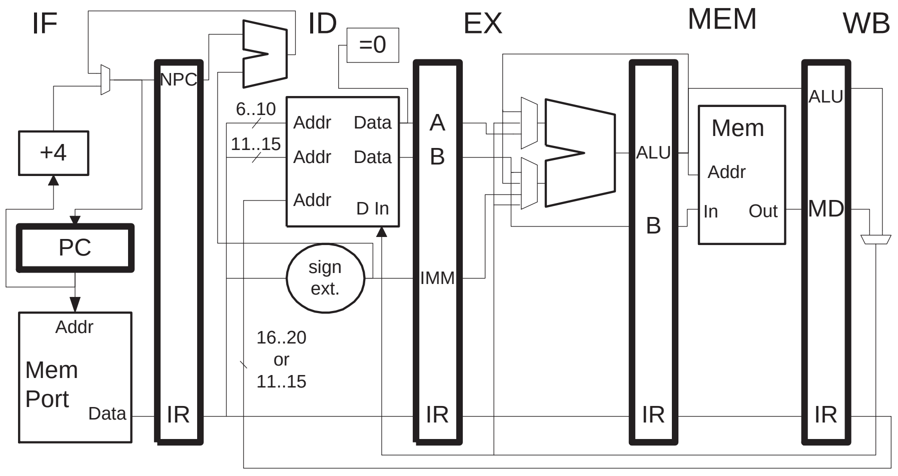
```
 ! Initially, r1=0x11, r2=0x22, r3=0x33, etc.
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, r0, r0
START: ! START = 0x50
 addi r1, r2, #1
 add  r2, r1, r6
 xor  r2, r1, r2
 bneq r1, START
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, 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 `add` is in instruction fetch. The first two columns are completed, filling up the rest of the table. Ignore values which are not used *and* which depend on the func field of type-R instructions. Values which are not used and don't depend on the func field should be shown. Don't forget the `IMM` values for `bneq`. 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.

Cycle	0	1	2	3	4	5	6	7	8	9	10
PC	0X50	0X54									
IF/ID.IR	sub	addi									
Reg. Chng.	r0 ← 0	r0 ← 0									
ID/EX.IR	sub	sub									
ID/EX.A	0	0									
ID/EX.B	0	0									
ID/EX.IMM	0	0									
EX/MEM.IR	sub	sub									
EX/MEM.ALU	0	0									
EX/MEM.B	0	0									
MEM/WB.IR	sub	sub									
MEM/WB.ALU	0	0									
MEM/WB.MD	0	0									

Problem 3: The program below executes on the implementation also shown below.



```
 add  r1, r2, r3
 and  r4, r1, r5
 sw   0(r4), r1
 lw   r1, 8(r4)
 xori r5, r1, #1
 beqz r5, TARGET
 sub  r5, r5, r5
 ...
TARGET:
 or   r10, r5, r1
```

The implementation includes only the forwarding paths that are shown in the figure. A new register value can be read in the same cycle it is written. Show a pipeline execution diagram for an execution of the code in which the branch is taken.

Problem 4: Add exactly those forwarding paths (but no others) that are needed in the DLX implementation used in the problem above so that the code above executes as quickly as possible. Show a pipeline execution diagram of the code (repeated below) on the modified implementation.

```
 add  r1, r2, r3
 and  r4, r1, r5
 sw   0(r4), r1
 lw   r1, 8(r4)
 xori r5, r1, #1
 beqz r5, TARGET
 sub  r5, r5, r5
 ...
TARGET:
 or   r10, r5, r1
```

Problem 5: The code below executes on the DLX implementation shown below which also includes the following floating-point hardware:

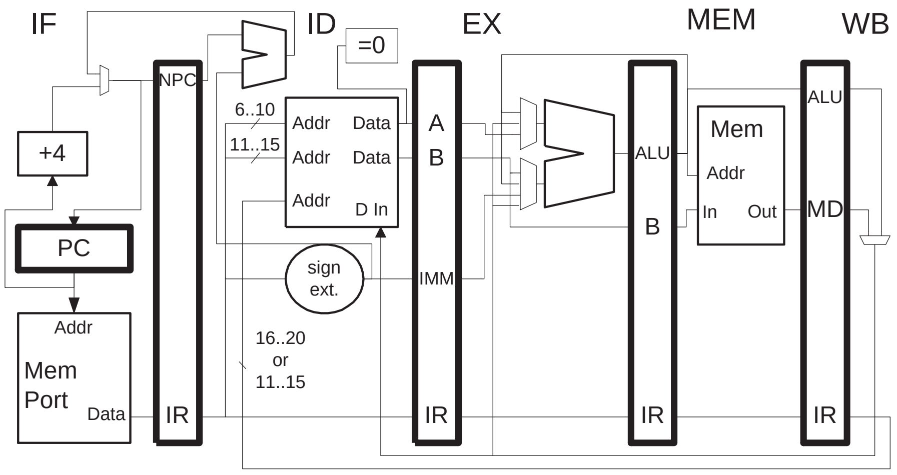
- As described in Section 3.7 of the text and in class, there is a four-stage FP add unit, a seven-stage multiply unit, and a 25-cycle FP divide unit (not used in the code below). The FP add unit also performs FP comparisons, such as `eqf`.
- The floating-point branch instructions, `bfpt` and `bfpf`, are executed in the ID stage just as the integer branches, `beqz` and `bneq`. The FP condition code register (also not shown) is updated in the WB cycle but the value to be written is forwarded to the controller at the beginning of WB.
- All stalls are in the ID stage. Floating-point instructions **skip** the MEM stage.
- Floating-point values are forwarded from the WB stage to the inputs of the FP execution units. A value written to a FP register can be read in the same cycle.

(*a*) Show a pipeline execution diagram for two iterations of the code below in which `bfpt` is taken in the first iteration but not taken in the second. (Note: the loop is infinite.)

(*b*) Determine the CPI of an execution of the code for a large number of iterations in which `bfpt` is always taken.

(*c*) Determine the CPI of an execution of the code for a large number of iterations in which `bfpt` is never taken.

(*d*) Determine the CPI of an execution of the code for a large number of iterations in which `bfpt` is taken 50% of the time.



```
LOOP:
 addi  r1, r1, #8
 lf    f0, 0(r1)
 addf  f1, f1, f0
 eqf   f0, f2
 bfpt  LOOP
 multf f1, f1, f3
 beqz  r0, LOOP
 xor   r2, r1, r3
```