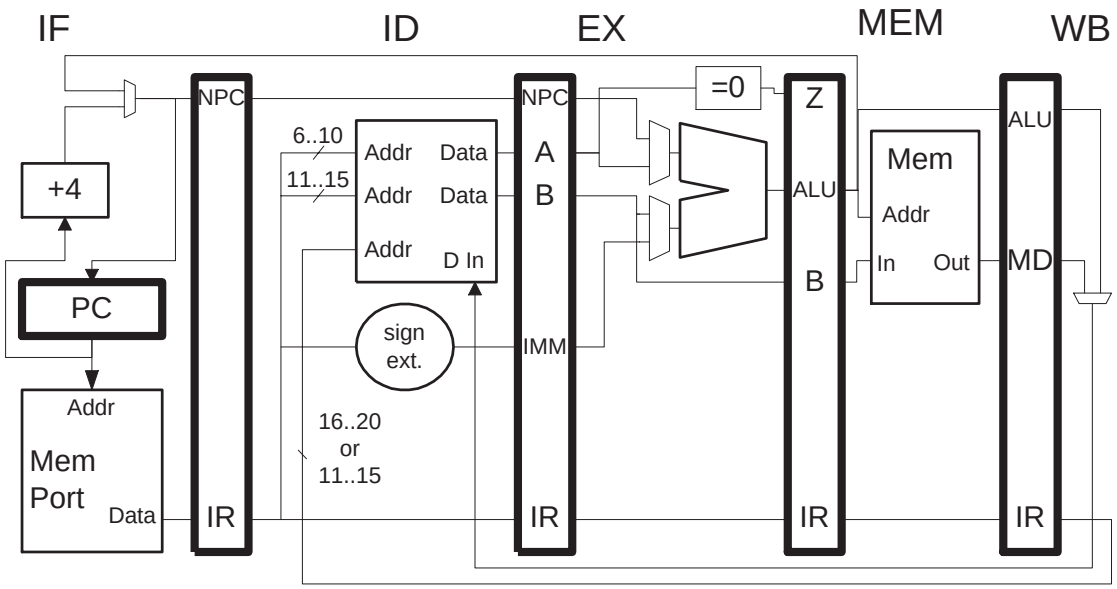
EE 4720 **Homework 3** **Due: 8 March 1999**

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.)*

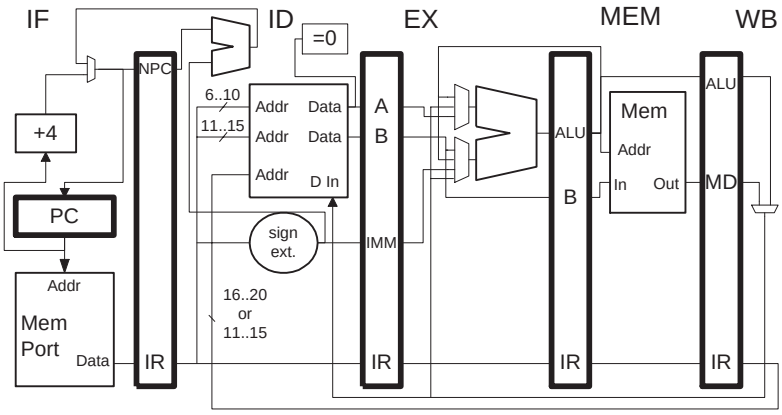


The problem: The `jal` and `jalr` instructions are supposed to save the return address (`NPC`) in `r31` but in the pipeline above there is no path that `NPC` can take to the writeback stage. (The path through the ALU could be used if it wasn't already being used to compute the target address.)

The solution: provide `EX/MEM.NPC` and `MEM/WB.NPC` pipeline latches and connect them so that the return address can move to the writeback stage without having to go through the ALU. Connect the output of `MEM/WB.NPC` to the multiplexor leading to the register file. (`MEM/WB.ALU` and `MEM/WB.MD` are already connected to this multiplexor.)

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`.

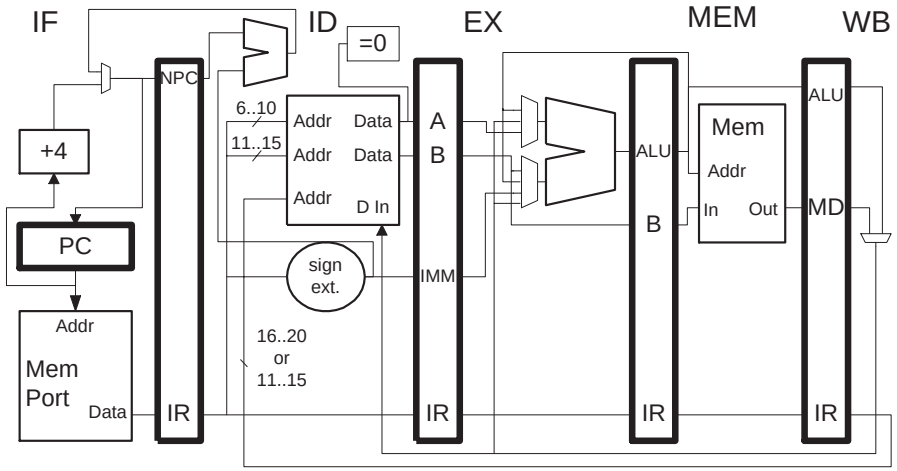
```
 ! Initially, r1=0x11, r2=0x22, r3=0x33, etc.
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, r0, r0
START: ! START = 0x50
 addi r1, r2, #1
 add  r2, r1, r6
 xor  r2, r1, r2
 bneq r1, START
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, r0, r0
 sub  r0, 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 `addi` is in instruction fetch. The first two columns are completed; fill 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	0x58	0x5c	0x60	0x50	0x54	0x58	0x5c	0x60	0x50
IF/ID.IR	sub	addi	add	xor	bneq	sub	addi	add	xor	bneq	sub
Reg. Chng.	r0 ←0x0	r0 ←0x0	r0 ←0x0	r0 ←0x0	r1 ←0x23	r2 ←0x89	r2 ←0xaa	x	r0 ←0x0	r1 ←0xab	r2 ←0x111
ID/EX.IR	sub	sub	addi	add	xor	bneq	or	addi	add	xor	bneq
ID/EX.A	0x0	0x0	0x22	0x11	0x11	0x23	0x0	0xaa	0x23	0x23	0xab
ID/EX.B	0x0	0x0	0x11	0x66	0x22	0x0	0x0	0x23	0x66	ab	0x0
ID/EX.IMM	0x0	0x0	0x1	?	?	-0x10	0x0	0x1	?	?	0x14
EX/MEM.IR	sub	sub	sub	addi	add	xor	bneq	or	addi	add	xor
EX/MEM.ALU	0x0	0x0	0x0	0x23	0x89	0xaa	?	0x0	0xab	0x111	0x1ba
EX/MEM.B	0x0	0x0	0x0	0x11	0x66	0x22	0x0	0x0	0x23	0x66	0xaa
MEM/WB.IR	sub	sub	sub	sub	addi	add	xor	bneq	or	addi	add
MEM/WB.ALU	0x0	0x0	0x0	0x0	0x23	0x89	0xaa	?	0x0	0xab	0x111
MEM/WB.MD	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

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.

Solution:

```
Cycle:            0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
 add  r1, r2, r3  IF  ID  EX  MEM WB
 and  r4, r1, r5      IF  ID  EX  MEM WB
 sw   0(r4), r1           IF  ID  --> EX  MEM WB
 lw   r1, 8(r4)               IF  --> ID  EX  MEM WB
 xori r5, r1, #1                  IF  ID  --> EX  MEM WB
 beqz r5, TARGET                      IF  --> ID  -----> EX  MEM WB
 sub  r5, r5, r5                          IF  -----> x
 ...
TARGET:
 or   r10, r5, r1                                        IF  ID  EX  MEM WB
```

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
```

The execution in the previous problem suffers three stalls, starting at cycles 4, 7, and 9.

Without the stall at cycle 4 there would be no way for the data (the new value of `r1`) to reach the `EX/MEM.B` pipeline latch when `sw` is at the MEM stage. This can be fixed with a bypass connection from the output of the writeback-stage multiplexor to a new multiplexor placed at the inputs to the `EX/MEM.B` pipeline latch.

The stall at cycle 7 cannot be avoided since the data is first available at the end of cycle 7 but would be needed at the beginning of cycle 7 (if the stall were removed).

The stall at cycle 9 provides time for the new value of `r5` to reach WB where it meets `beqz` at cycle 10. One or both stall cycles can be eliminated by inserting bypass paths. To eliminate one stall cycle insert a bypass path from `EX/MEM.ALU` to the input of the `=0` box in ID. To eliminate both stall cycles (while possibly lengthening the critical path) insert a bypass path from the ALU output (before the `EX/MEM` pipeline latch) to the `=0` box.

The pipeline execution diagram below uses the conservative approach for the `=0` bypass, from `EX/MEM.ALU`:

```
Cycle:            0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
 add  r1, r2, r3  IF  ID  EX  MEM WB
 and  r4, r1, r5      IF  ID  EX  MEM WB
 sw   0(r4), r1           IF  ID  EX  MEM WB
 lw   r1, 8(r4)               IF  ID  EX  MEM WB
 xori r5, r1, #1                  IF  ID  --> EX  MEM WB
 beqz r5, TARGET                      IF  --> ID  --> EX  MEM WB
 sub  r5, r5, r5                              IF  --> x
 ...
TARGET:
 or   r10, r5, r1                                     IF  ID  EX  MEM WB
```

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

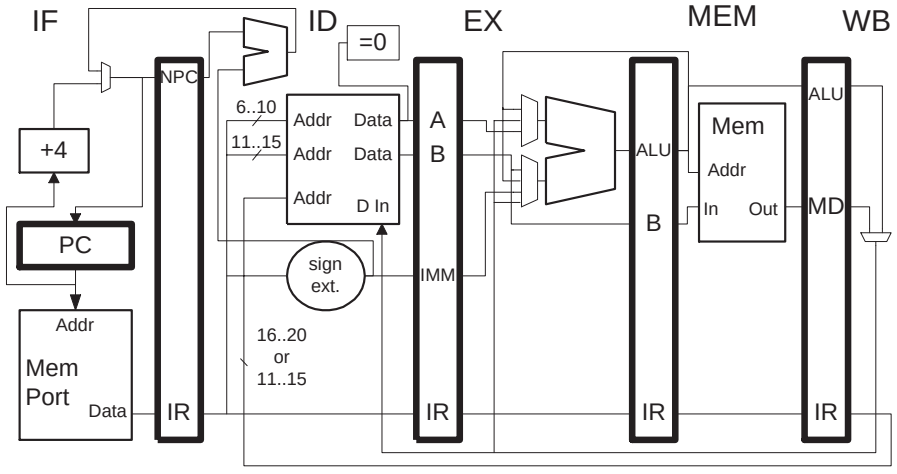
- As described in Section 3.7 of the text and in class, there is a four-stage FP add unit, a seven-stage multiply unit, and a 25-cycle FP divide unit (not used in the code below). The FP add unit also performs FP comparisons, such as `eqf`.
- The floating-point branch instructions, `bfpt` and `bfpf`, 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
```

Solution: (The label for the memory (MEM) stage has been shortened to ME. Three iterations (rather than two) are shown; they are needed to solve part (c).)

```
LOOP:
 Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
 addi  r1, r1, #8 IF ID EX ME WB                   IF ID EX ME WB                         IF ID EX ME WB                            IF ...
 lf    f0, 0(r1)     IF ID EX ME WB                   IF ID EX ME WB                         IF ID EX ME WB
 addf  f1, f1, f0       IF ID -> A0 A1 A2 A3 WB          IF ID -> A0 A1 A2 A3 WB                IF ID ----> A0 A1 A2 A3 WB
 eqf   f0, f2              IF -> ID A0 A1 A2 A3 WB          IF -> ID A0 A1 A2 A3 WB                IF ----> ID A0 A1 A2 A3 WB
 bfpt  LOOP                     IF ID ----------> EX ME WB       IF ID ----------> EX ME WB                IF ID ----------> EX ME WB ...
 multf f1, f1, f3                  IF ----------> x                 IF ----------> ID M0 M1 M2 M3 M4 M5 M6 WB IF ----------> ID M0 M1 ...
 beqz  r0, LOOP                                                                    IF ID EX ME WB                            IF ID EX ...
 xor   r2, r1, r3                                                                     IF x                                      IF x
```

Part (b): If **bfpt** is taken the iteration consists of 5 instructions. If the branch is always taken each iteration will execute as the first above, and so there will be 11 cycles per iteration. The CPI is $11/5 = 2.2\,\mathrm{CPI}$.

Part (c): If **bfpt** is not taken the iteration consists of 7 instructions. In the second iteration above the branch is not taken and so **multf** is executed, producing a new value of **f1**. That new value is needed in the third iteration, stalling **addf** an extra cycle (the stall occurs in cycles 28 and 29). The second iteration takes 13 cycles (from cycle 11 to 24) but due to the extra cycle the third iteration takes 14 cycles (from cycle 24 to 38). Because iteration 3 and 4 start the same way (as can be determined by examining the state of execution [a vertical strip] at cycles 24 and 38) they should take the same number of cycles as should following iterations as long as the branch is not taken. (Note that iteration 2 at cycle 11 starts differently.) Therefore the CPI is $14/7 = 2\,\mathrm{CPI}$.

Part (d): An iteration where **bfpt** is taken that follows an iteration where it isn't would take 12 cycles (such a pair is not shown in the diagram above). An iteration where **bfpt** is not taken that follows an iteration where it is would take 13 cycles; for example, the second iteration above. For a large number of iterations the CPI would be $(12 + 13)/(5 + 7) = 2.083\,\mathrm{CPI}$.