A few words about the quiz

- Closed book, but you may bring in a page of handwritten notes.
  - You need to know what the "core" MIPS instructions do.
  - I don't really think of you'll need notes, but it's up to you.
- In "fill in the blank" questions, you will get partial credit for answering "I don't know" or leaving it blank. (For instance, if there are 3 choices, you will get 1/3 credit for leaving it blank.)
- The relevant circuit diagrams will be in the test.
- Pages 3 and 4 are exercises that may be similar to test question. Answers are given on pages 6 and 7.

Some information about the MIPS

This information will be restated on the test, but it might save you time to read it over beforehand.

- Some "core" MIPS assembly instructions:
  - lw = load word
  - sw = store word
  - add, sub, and, or, xor = various R-format instructions
  - beq = branch on equal
  - j = jump
- The first argument of MIPS assembly instructions is the target register (except for sw). Note that the arguments don't appear in the same order as in the machine language instruction format.
- MemRead = 0 means "don't read", MemRead = 1 means "read". Similarly, MemWrite=0 means "don't write", MemWrite = 1 means "write".
- The two-bit ALUop signal, which goes from the "control" unit to the "ALU control" unit, has the following possible values:
  - "00" means "the ALU should add",
  - "01" means "the ALU should test for equality"
  - "10" means "the ALU should execute the operation specified by the funct field (bits 0-5) of the instruction"
- The mux control signals select the inputs as shown in the diagrams. They may be different from the book or lectures – use what is shown on the test diagrams.

Pipelining Exercises

Consider the following MIPS assembly code:

```
add $3, $2, $3
lw $4, 100($3)
sub $7, $6, $2
xor $6, $4, $3
```

Assume there is no forwarding or stalling circuitry in a pipelined processor that uses the standard 5-stages (IF, ID, EX, Mem, WB). Instead, we will require the compiler to add no-ops to the code to ensure correct execution. (Assume that if the processor reads and writes to the same register in a given cycle, the value read out will be the new value that is written in.)

1. Rewrite the code to include the no-ops that are needed. Do not change the order of the four statements. Use as few no-ops as possible.

2. Suppose the complier is allowed to change the order of the four statements, provided it doesn't change the final answer. Is it possible to reduce the number of no-ops needed? Why or why not?

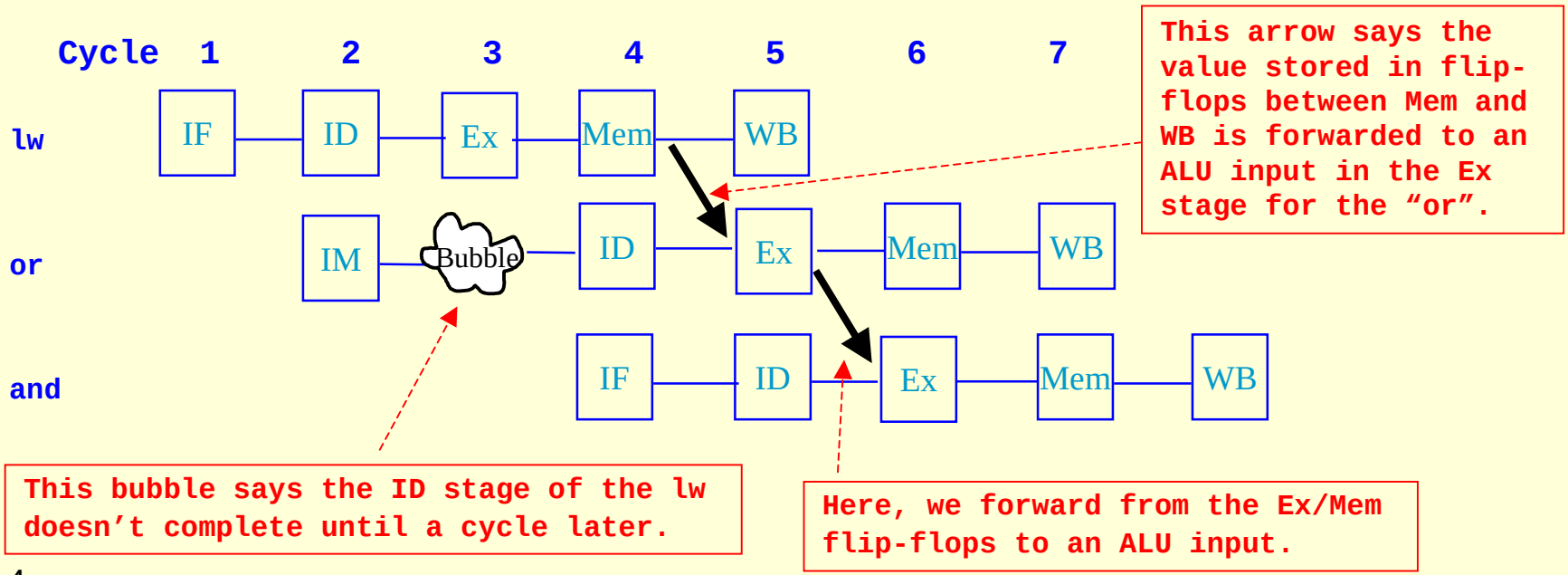
Another pipelining exercise

Consider (again) the following MIPS assembly code:

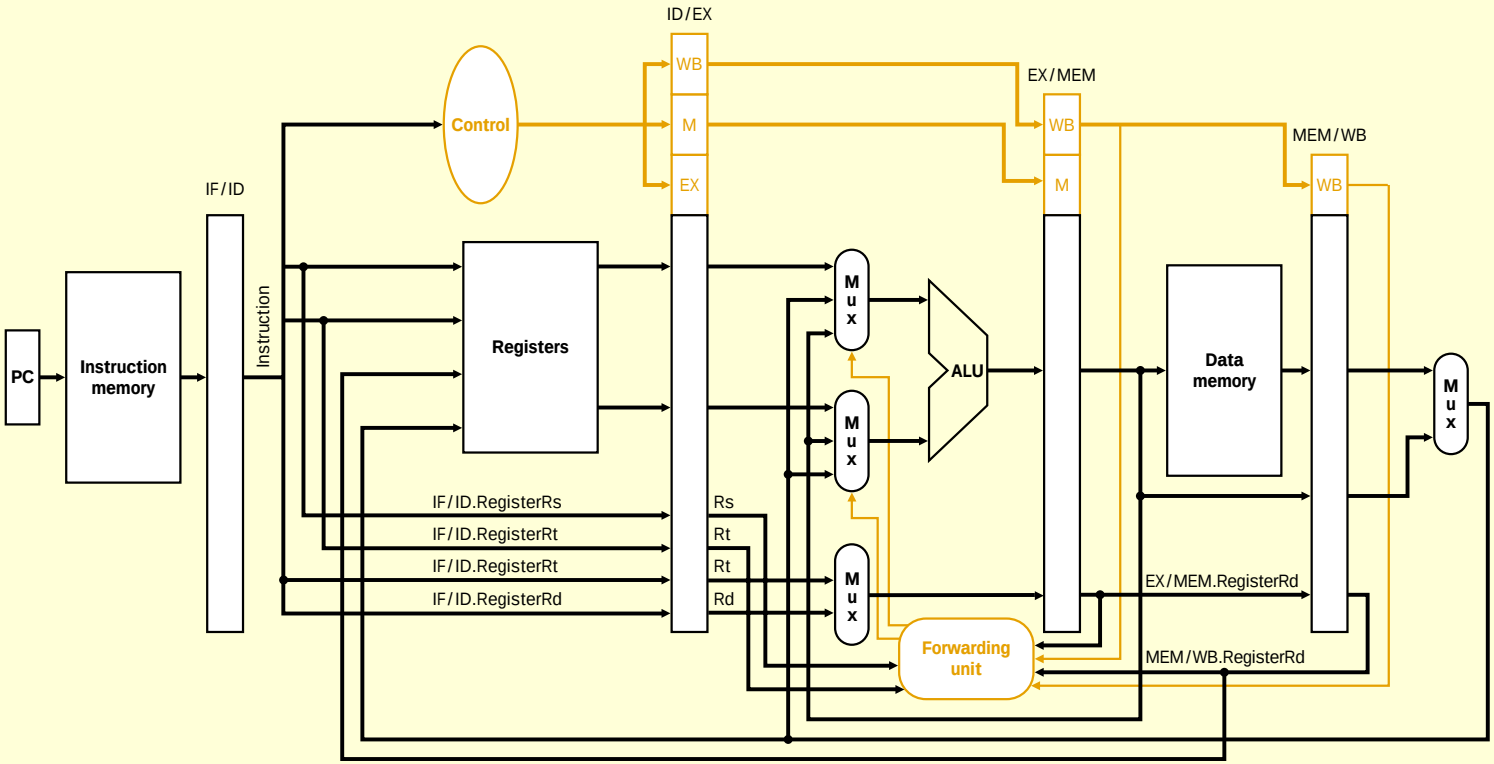
```
add $3, $2, $3
lw $4, 100($3)
sub $7, $6, $2
xor $6, $4, $3
```

Assume that there is forwarding and stalling hardware as described in class and the text (the forwarding path is shown in the diagram on the next page).

Draw an execution diagram that shows where forwarding and stalling would take place. Use arrows to show forwarding and "bubbles" to show stalls, as in the following (hypothetical) example.



The pipelined implementation with forwarding (used in the question on the previous page)



Solution to 1 & 2

1. Rewrite the code to include the no-ops that are needed. Do not change the order of the four statements. Use as few no-ops as possible.

```
add $3, $2, $3
no-op
no-op
lw $4, 100($3)
sub $7, $6, $2
no-op
xor $6, $4, $3
```

The new value of register 3 doesn't reach the register file until the WB stage of the add instruction, i.e. cycle 5. But the lw instruction needs this value in its ID stage, when it reads register 3 To delay the ID stage of the lw until cycle 5, we need to insert 2 no-ops.

The new value of register 4 doesn't reach the register file until the WB stage of the lw instruction. If we didn't insert this no-op, the ID stage of the xor would coincide with the Mem stage of the lw.

If this were a test, you wouldn't need to write these explanations; what matters is inserting the no-ops correctly.

2. Suppose the complier is allowed to change the order of the four statements, provided it doesn't change the values that are computed. Is it possible to reduce the number of no-ops needed? Why or why not?

No. Even though the "sub" instruction can be moved earlier, the "lw" can't start until three cycles after the "add" starts, and the "xor" can't start until three cycles after that. Thus, the "xor" can't start until 6 cycles after the add.

Solution to “Another pipelining exercise”

Consider (again) the following MIPS assembly code:

```
add $3, $2, $3
lw $4, 100($3)
sub $7, $6, $2
xor $6, $4, $3
```

Draw an execution diagram that shows where forwarding and stalling would take place.