Name _____

Computer Architecture
EE 4720
Midterm Examination

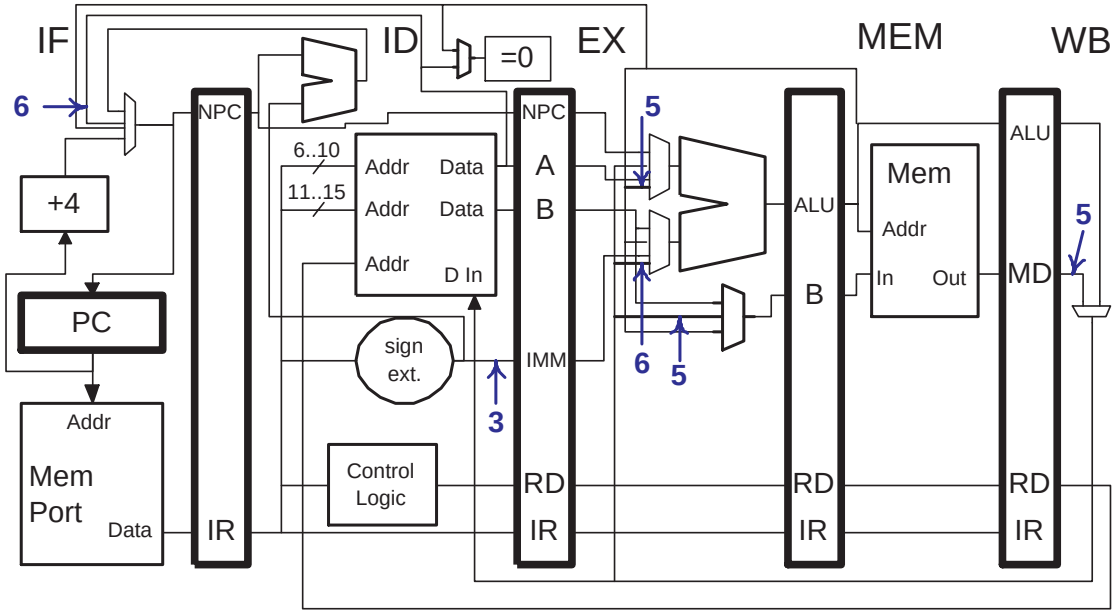
Wednesday, 21 March 2001, 13:40–14:30 CST

Problem 1	_____	(35 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(40 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: In the DLX implementation below six paths are marked with a number, a cycle in which the path will be used. Write a DLX program that uses those six paths at the indicated cycles. Some marked paths are *bypass* paths and some are not; the bypass paths can be used **only** at the cycles indicated. A pipeline execution diagram is provided for your convenience.



☐ [15 pts] Choose the register operands so the bypass paths are used **only** at the cycles indicated.

☐ [15 pts] Choose the instructions so that the marked paths will be used as indicated. There need be only one control transfer instruction.

☐ [5 pts] One instruction will be squashed. Show where by crossing out the segment labels (`ID`, etc.) in the diagram.

```
! Cycle     0   1   2   3   4   5   6   7   8   9   10  11  12
            IF  ID  EX  MEM WB
                IF  ID  EX  MEM WB
                    IF  ID  EX  MEM WB
                        IF  ID  EX  MEM WB
                            IF  ID  EX  MEM WB
                                IF  ID  EX  MEM WB
                                    IF  ID  EX  MEM WB
                                        IF  ID  EX  MEM WB
                                            IF  ID  EX  MEM WB
! Cycle     0   1   2   3   4   5   6   7   8   9   10  11  12
```

Problem 2: The DLX code below runs on a dynamically scheduled system that uses reorder buffer entries to name destination registers (Method 1). The floating-point adder has **five stages**, `A1` through `A5`, and is fully pipelined. The common data bus (CDB) can handle an unlimited number of writebacks per cycle, but other parts of the implementation are ordinary. The cache is non-blocking. The `cvtftoi` instruction uses the FP adder; the `movfptoi` instruction uses the integer (`EX`) functional unit. The pipeline is fully bypassed.

(*a*) For this part no instructions raise exceptions.

☐ [7 pts] Show a pipeline execution diagram for the code. Reorder buffer and reservation station numbers **do not** have to be shown.

☐ [5 pts] Indicate where each instruction commits.

Check the code carefully for dependencies! Register `r1` *is part of a long chain of dependencies. Pay attention to the register equality and inequality comment.*

```
!  r4 = r2,  r6 != r2,  r6 != r1
!Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16 17 18 19
addf f0, f1, f2

cvtftoi f0, f0

movfptoi r1, f0

subd f0, f4, f6

sd 0(r1), f10

sd 0(r2), f0

ld f12, 0(r4)

ld f14, 0(r6)

!Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16 17 18 19
```

(*b*) Suppose an arithmetic exception is discovered for the `subd` instruction when it is in the first FP adder stage, `A1`, in an execution of the program above.

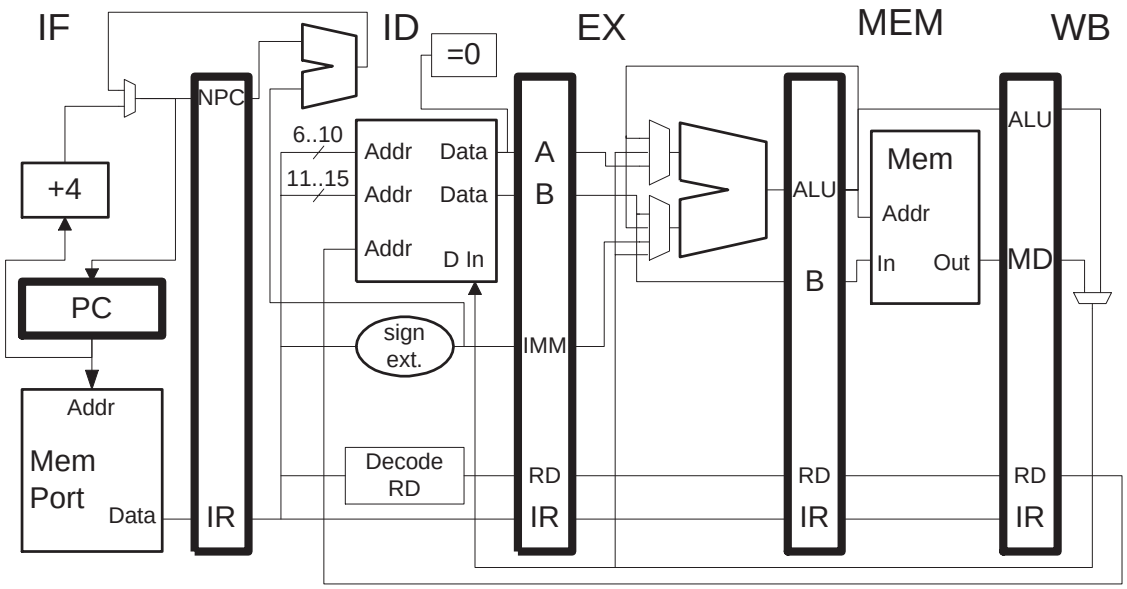
☐ [6 pts] At what cycle is the reorder buffer flushed and the handler fetched?

☐ [7 pts] Show the contents of the reorder buffer when the status of the `subd` instruction is set to exception. (For partial credit show the contents of the reorder buffer halfway through execution, be sure to indicate the cycle.)

Problem 3: Answer each question below.

(*a*)

☐ [8 pts] Design the control logic for the WB-stage multiplexor. The control logic itself must be in the ID stage. Show the connection to the mux and number the mux inputs. *Hint: This is an easy problem.*



(*b*) In the pipeline execution diagram below `multd` is delayed because of a true dependency with the `addd` instruction.

☐ [8 pts] Why can't any realistic implementation do things this way?

```
addd  f0, f2, f4   IF ID A1 A2 A3 A4 WB
multd f6, f0, f8               IF ID M1 M2 M3 M4 M5 M6 WB
subd  f10, f12, f14               IF ID A1 A2 A3 A4 WB
```

(*c*) Unlike DLX, many ISAs, such as SPARC, use a condition code register for integer branch conditions.

☐ [4 pts] In what way is DLX's method more flexible?

☐ [4 pts] In what way is a condition code register more flexible?

(*d*) An ISA may have variable-width instructions, fixed-width instructions, and bundled instructions.

☐ [4 pts] How do the different alternatives affect displacement branches?

[4 pts] Name an ISA category (type) that uses each instruction format:

☐ Variable-Width Instructions:

☐ Fixed-Width Instructions:

☐ Bundled Instructions:

(*e*) Packed-operand data types and instructions are *de rigueur* for any *au courant fin-de-siècle* ISA. (Are a must-have for any up-to-date end-of-the-century ISA.)

☐ [8 pts] What are packed-operand data types and instructions? Show a short program that would benefit from these. The program can be in a high-level language or even pseudo code. Do not show the packed-operand instructions, just explain what they would do.