Name ____________________

Computer Architecture
EE 4720
Midterm Examination

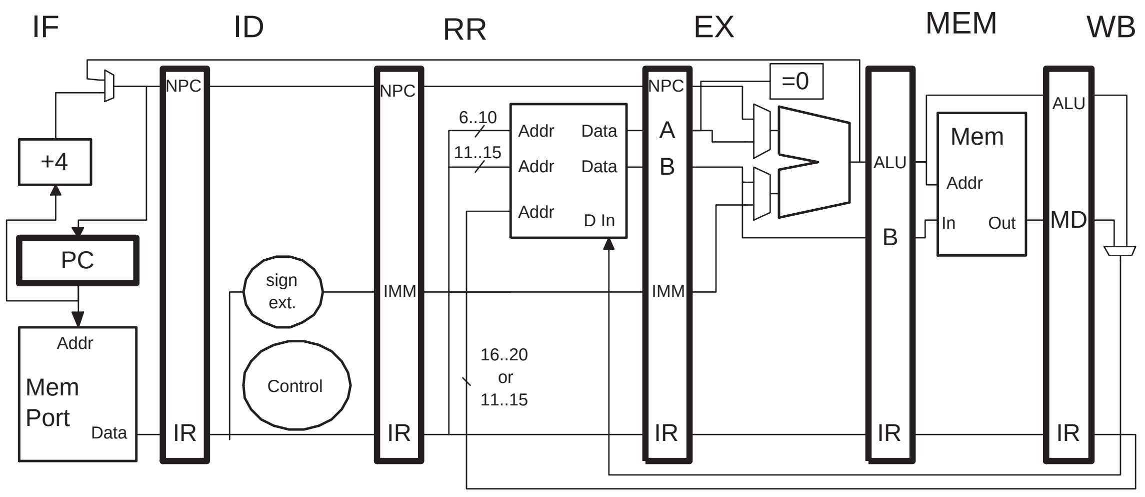
22 March 2000, 13:40–14:30 CST

Problem 1	______	(35 pts)
Problem 2	______	(20 pts)
Problem 3	______	(45 pts)
Exam Total	______	(100 pts)

Alias ____________________

Good Luck!

Problem 1: The DLX implementation below has six stages. (The work done by ID is now done by ID and RR.)



(*a*) The execution of some code on this pipeline is shown below. Add *exactly* the bypass paths needed so that the code executes as illustrated. Next to each bypass path indicate the cycle(s) in which it will be used. (Do not add bypass paths that won't be used in the execution of the code below.) (10 pts)

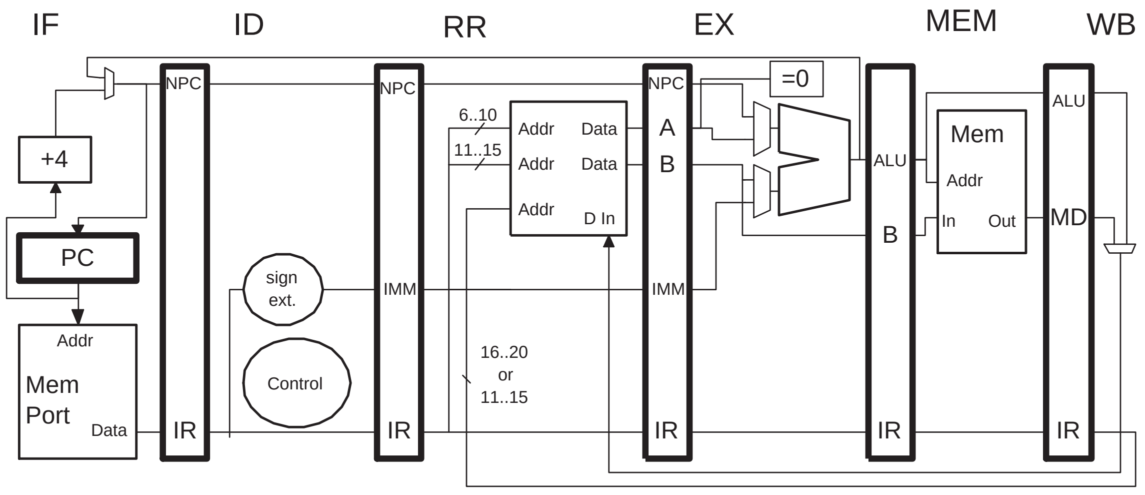
```
 ! Cycle          0  1  2  3  4  5  6  7  8
 add  r1, r2, r3  IF ID RR EX ME WB
 lw   r5, 10(r1)     IF ID RR EX ME WB
 xor  r4, r1, r2        IF ID RR EX ME WB
 and  r6, r5, r4           IF ID RR EX ME WB
```

(*b*) Show the execution of the code below on this pipeline until `bneq` reaches IF a second time. The branch is taken. Be sure to base the CTI behavior on the hardware shown above. Show where instructions are squashed. (10 pts)

```
LOOP:
 ! Branch Taken.
 bneq r1, SKIP
 add  r2, r3, r4
 sub  r5, r6, r7
 and  r8, r9, r10
 or   r11, r12, r13
SKIP:
 j LOOP
 add  r2, r3, r4
 sub  r5, r6, r7
 and  r8, r9, r10
 or   r11, r12, r13
```

Problem 1, continued: The figure and code below are identical to the ones on the previous page.

(*c*) Add branch and jump hardware so that the code executes as quickly as possible. Additional adders can be used, the fewer the better. Branches and jumps can be handled separately. The register file cannot be moved or duplicated and cannot be read before the RR stage. (Jumps and branches both use displacement addressing. In branches the displacement is in bits 16 to 31 and in jumps the displacement is in bits 6 to 31. Do not show control hardware.) (10 pts)



(*d*) Show how the code below executes on the modified pipeline. As before, show execution until `bneq` enters IF a second time. (5 pts)

```
LOOP:
 ! Branch Taken.
 bneq r1, SKIP
 add  r2, r3, r4
 sub  r5, r6, r7
 and  r8, r9, r10
 or   r11, r12, r13
SKIP:
 j LOOP
 add  r2, r3, r4
 sub  r5, r6, r7
 and  r8, r9, r10
 or   r11, r12, r13
```

Problem 2: The code below executes on a dynamically scheduled processor that uses reorder buffer entry numbers to name registers. There are 128 reorder buffer entries, the next free entry is #1.

(*a*) Show when each instruction commits and show the contents of the register map, register file and reorder buffers using the space provided. Show only the instruction line numbers (`A`, `B`, `C`, `D`) in the reorder buffer. Indicate cycle numbers above the reorder buffer. (10 pts)

Pipeline Execution Diagram

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`A:addd f0, f2, f0`	IF	ID	A0	A1	A2	A3	WB										
`B:addd f6, f0, f6`		IF	ID	RS	RS	RS	A0	A1	A2	A3	WB						
`C:addd f0, f0, f6`			IF	ID	RS	RS	RS	RS	RS	RS	A0	A1	A2	A3	WB		
`D:addd f6, f2, f8`				IF	ID	A0	A1	A2	A3	WB							

Register Map

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val. or ROB#																
`f0`	10																
`f2`	20																
`f6`	60																
`f8`	80																

Register File

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val.																
`f0`	10																
`f2`	20																
`f6`	60																
`f8`	80																

Cycle:

Reorder buffer:

Problem 2, continued: (*b*) The code below is identical to the code above, however the second add instruction raises an exception in cycle 9 (indicated by a star). Complete the pipeline execution diagram and other tables for this situation for the instructions shown. (Do not show the trap handler.) Show the contents of the reorder buffers after each change. (10 pts)

Pipeline Execution Diagram

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`A:addd f0, f2, f0`	IF	ID	A0	A1	A2	A3	WB										
`B:addd f6, f0, f6`		IF	ID	RS	RS	RS	A0	A1	A2	*A3*							
`C:addd f0, f0, f6`			IF	ID	RS	RS	RS	RS	RS	RS							
`D:addd f6, f2, f8`				IF	ID	A0	A1	A2	A3	WB							

Register Map

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val. or ROB#																
f0	10																
f2	20																
f6	60																
f8	80																

Register File

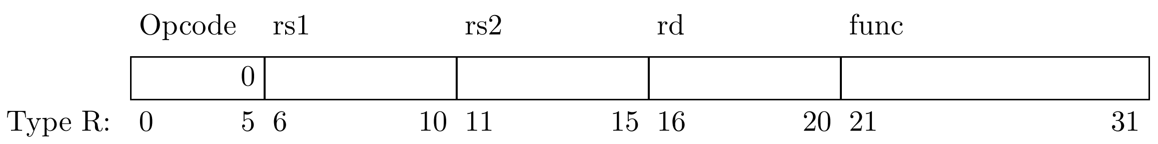
Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val.																
f0	10																
f2	20																
f6	60																
f8	80																

Cycle:

Reorder buffer:

Problem 2: Answer each question below.

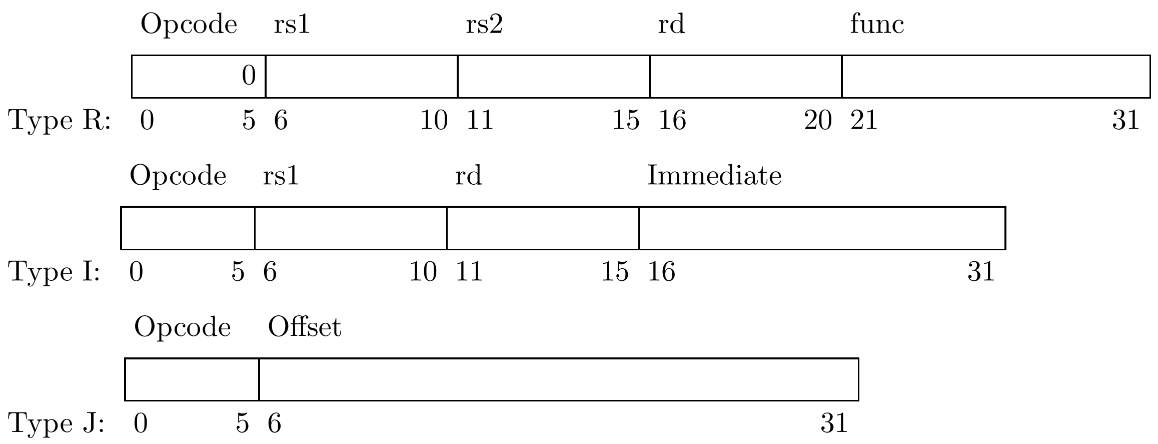
(*a*) Why do DLX Type-R instructions have a `func` field? (7 pts)



(*b*) Would there be any advantage in pipelining the integer ALU used in the statically scheduled DLX implementation? Explain. (7 pts)

(*c*) Why do ISAs include one-byte integers? (What are the advantages of using them when running on typical implementations.) (7 pts) (*d*) The DLX implementations covered in class start reading registers before the instruction is decoded. Why is this possible? Modify the instruction codings so that it is no longer possible. That is, with the modified codings decoding would have to be performed *before* register read (as in problem 1). (8 pts)

The DLX codings are given below for reference and can be used to explain your answer.



(*e*) Ignoring floating-point instructions, how can precise exceptions be implemented on the statically scheduled (Chapter 3) DLX implementation? Illustrate your answer with an example in which exceptions occur out of order but are handled in order. Show the code and a pipeline execution diagram, indicate where the exceptions are occurring. (9 pts)

(*f*) What is an advantage of a stack ISA over a RISC ISA? What is a disadvantage of a stack ISA over a RISC ISA? (7 pts)