Name _____

Computer Architecture
EE 4720
Midterm Examination

12 March 1999, 10:40–11:30 CST

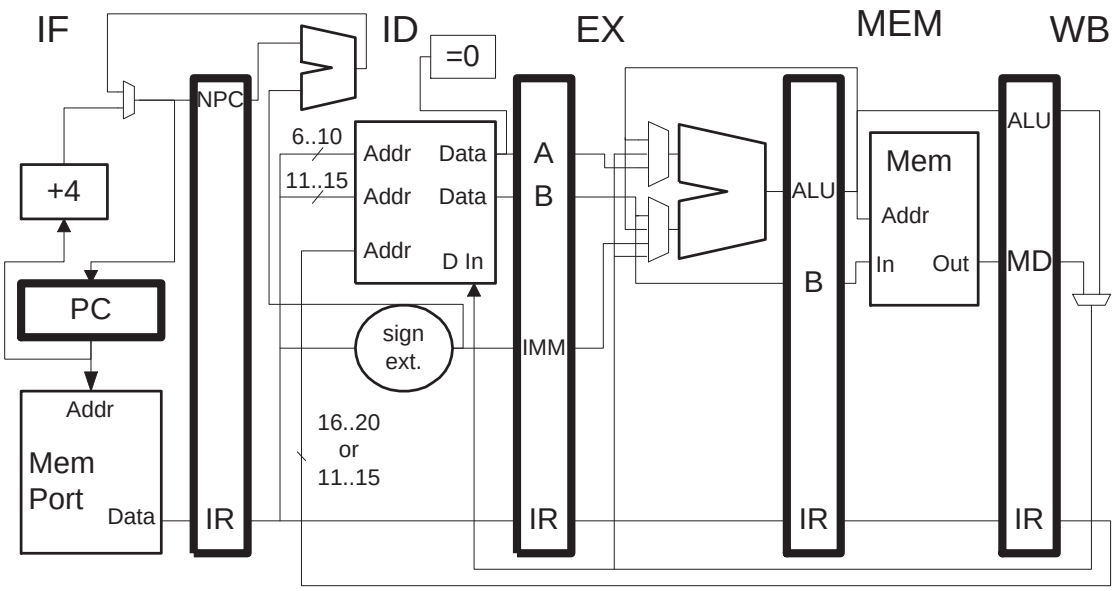
Preliminary, Partial Solutions

Problem 1	_____	(25 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(45 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: Design control logic to generate the control signal for the multiplexor at the lower input to the ALU. The control logic should be located in the ID stage and should generate a two-bit integer for the multiplexor. The integer specifies which multiplexor input to use, they are numbered from zero starting at the top. (Input 0 connects to `ID/EX.B`, 1 connects to `EX/MEM.ALU`, etc.) The logic can use units that test for equality of their two inputs, `=`, and units that test for instruction formats, `= Type I`, `= Type R`, and `= Type J`, and can use the usual logic gates. Base the setting on instruction type, rather than the exact opcode. Show how the control signal is connected to the multiplexor. (25 pts)



Problem 2: Consider the code below.

```
LOOP:
Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
 lw   r1, 0(r2)    IF  ID  EX  MEM WB                                  IF  ID  EX
 addi r3, r1, #12      IF  ID  --> EX  MEM WB                              IF  ID
 sw   4(r2), r3            IF  --> ID  ----->  EX  MEM WB                      IF
 add  r5, r5, r1                   IF  ----->  ID  EX  MEM WB
 addi r2, r2, #8                               IF  ID  EX  MEM WB
 slt  r6, r2, r7                                   IF  ID  EX  MEM WB
 bneq r6, LOOP                                         IF  ID  ----->  EX  MEM WB
 xor  r8, r9, r10                                          IF  ----->  x
```

(a) Show a pipeline execution diagram for execution up to the second time `lw` enters instruction fetch. Use the pipeline from problem 1. As with homework 3, a bypass path is unavailable if it's not shown. What is the CPI for a large number of iterations? (10 pts)

The pipeline execution diagram appears above. The CPI is $\frac{13}{7} = 1.857$ CPI.

(b) Unroll the loop so that two iterations of the code above is performed by one iteration of the unrolled loop. (Assume the number of iterations in the original loop is a multiple of two.) Schedule the unrolled loop to minimize stalls. (10 pts)

```
LOOP:
!Cycle:            0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
 lw   r1, 0(r2)    IF  ID  EX  MEM WB                                  IF  ID  EX  MEM
 lw   r11, 8(r2)       IF  ID  EX  MEM WB                                  IF  ID  EX
 addi r3, r1, #12          IF  ID  EX  MEM WB                                  IF  ID
 addi r13, r11, #12            IF  ID  EX  MEM WB                                  IF
 addi r2, r2, #16                  IF  ID  EX  MEM WB
 slt  r6, r2, r7                       IF  ID  EX  MEM WB
 add  r5, r5, r1                           IF  ID  EX  MEM WB
 add  r5, r5, r11                              IF  ID  EX  MEM WB
 sw   -12(r2), r3                                  IF  ID  EX  MEM WB
 sw   -4(r2), r13                                      IF  ID  EX  MEM WB
 bneq r6, LOOP                                             IF  ID  EX  MEM WB
 xor  r8, r9, r10                                              IF  x
```

(c) What is the CPI of the unrolled and scheduled loop found above? What conclusions about performance improvement can and cannot be made by comparing the CPI of the original and unrolled loop? What is the performance improvement? (Give a number for performance improvement, don't just say "it's good.") (10 pts)

The unrolled loop has 11 instructions and only suffers a 1-cycle branch delay, for a CPI of $\frac{12}{11} = 1.091$ CPI.

Though the CPI is lower, this doesn't tell the whole story because fewer instructions do the same amount of work and so the performance improvement is more than CPI improvement would suggest.

Performance improvement will be expressed as speedup. The speedup of the unrolled loop will be found using the time needed to do two iterations of the original loop and dividing it by the iteration time of the unrolled and scheduled loop: $\frac{13 \times 2}{12} = 2.167$.

Problem 3: Answer each question below.

(a) Show an example of DLX code that encounters a WAW hazard on the Chapter-3 implementation of DLX (in which the multiply floating-point functional unit has an initiation interval of 1 and a latency of 6 and the add floating-point functional unit has an initiation interval of 1 and a latency of 3) but which does not encounter a WAW hazard on a DLX implementation which is identical except the FP add latency is 1 and the FP multiply latency is 4. The code should not encounter a structural hazard on either implementation. (12 pts)

```
! Code execution on Chapter-3 DLX (unmodified).
! Cycle           0   1   2   3   4   5   6
addf  f0, f1, f2  IF  ID  A0  A1  A2  A3  WB
lf    f0, 0(r1)       IF  ID  EX  MEM WB

! Code execution on Chapter-3 DLX with fast FP functional units.
! Cycle           0   1   2   3   4   5   6
addf  f0, f1, f2  IF  ID  A0  A1  WB
lf    f0, 0(r1)       IF  ID  EX  MEM WB
```

(b) In DLX, why are there separate **lh** (load half) and **lhu** (load half unsigned) instructions, a **sh** (store half) instruction but **no shu** (store half unsigned) instruction? (11 pts)

Because the register contents is stored into a location of the right size and so there is no need for sign extension and therefore no need to distinguish a signed and unsigned value.

(*c*) The code below is for a stack architecture. Rewrite the code below in DLX using as few instructions as possible. Assume that `ADDRA` is in `r10`, `ADDRB` is in `r11`, `ADDRX` is in `r12`, and `ADDRY` is in `r13`. The data at the addresses are double-precision floating-point values. (11 pts)

```
push ADDRX
push ADDRX
mult
push ADDRA
push ADDRB
add
push ADDRX
mult
push ADDRA
push ADDRB
mult
add
add
pop ADDRY
```

```
ld    f0, 0(r10)  ! A
ld    f2, 0(r11)  ! B
ld    f4, 0(r12)  ! X
multd f6, f0, f0
addd  f8, f2, f4
multd f8, f8, f4
multd f10, f0, f2
addd  f10, f10, f8
addd  f10, f10, f6
sw    0(r13), f10
```

(*d*) Explain two ways in which precise exceptions can be made optional for floating-point instructions. That is, the programmer may choose to have precise exceptions where they are needed or may choose to not have precise exceptions were performance is most important. Explain what the programmer would have to do for each of the two ways. (11 pts)

Method 1: Provide precise and non-precise versions of floating-point instructions. The programmer uses the appropriate version.

Method 2: Provide a test instruction that can be used after floating-point instructions for which exceptions must be precise.