COS 471A,COS 471B/ELE 375 Midterm

Prof: David August
TAs : Jonathan Chang
Junwen Lai
Neil Vachharajani

Fall 2004

Please write your answers clearly in the space provided. For partial credit, show all work. State all assumptions. You have 1 hour and 20 minutes for this exam. This midterm is closed book. Only one two-sided, handwritten 8.5x11 sheet is allowed. Put your name on every page. Write out and sign the Honor Code pledge before turning in the test. *"I pledge my honor that I have not violated the Honor Code during this examination."*

Question	Score
1	20 /20
2	20 /20
3	24 /20
4	20 /20
Total	84 /80

Name:

Course (circle one): COS471B/ELE375 COS471A

Honor Code: Sol Ution

1 Binary Arithmetic

1.1 Consider 2's complement 4-bit signed integer addition. Overflow occurs whenever the sum of the two operands cannot be represented in the given format. Using at most **three** of the following signals, explain how to compute the overflow from the addition of two 2's complement 4-bit signed integers.

- The carry-in of the most significant bit. (C_I)
- The carry-out of the most significant bit. (C_O)
- The sign of the first operand. (S_1)
- The sign of the second operand. (S_2)
- The sign of the sum. (S_R)

S_1	S_2	S_R	Overflow
0	0	0	No
0	0	1	Yes
0	1	0	No
0	1	1	No
1	0	0	No
1	0	1	No
1	1	0	Yes
1	1	1	No

$$\text{Overflow} = \overline{(S_1 \oplus S_2)} \wedge (S_1 \oplus S_2)$$

As an optimization, the following also works:

$$\text{Overflow} = C_I \oplus C_O$$

This works by recognizing that C_O is always the operand signs, if the signs match. Further C_I is the sign of the result in these cases.

If the operand signs are the same the sum sign must match. If the signs are different, there is no overflow.

1.2 Define the WiMPY precision IEEE 754 floating point format to be:

$$\underbrace{X}_{\textit{Sign}}\ \underbrace{XXX}_{\textit{Exponent}}\ \underbrace{XXXX}_{\textit{Mantissa}}$$

where each 'X' represents one bit. Convert each of the following WiMPY floating point numbers to decimal:

1. 00000000

   sign = +
   value = 0
   +0

2. 11011010

   sign = 1 (negative)
   bias = $2^{(3-1)} - 1 = 3$
   exponent = 5
   mantissa = 1.1010
   value = −mantissa × $2^{(\text{exponent} - \text{bias})}$
   $= -1.1010 \times 2^{(5-3)} = -1.1010 \times 2^2 = 110.10 = -6.5$

3. 01110000

   sign = +
   exponent = 111 → ∞
   +∞

2 Dependence Detection

2.1 This question covers your understanding of dependencies between instructions. Using the code below, list all of the dependence types (RAW, WAR, WAW). List the dependencies in the respective table by writing the instruction numbers involved with the dependence. You may not need to fill the entire table.

```
I0:   r1 = r2 + r3;
I1:   r3 = r1 - r2;
I2:   r4 = r1 + r3;
I3:   r1 = r2 * r3;
```

RAW Dependence		WAR Dependence		WAW Dependence	
From Instr	To Instr	From Instr	To Instr	From Instr	To Instr
I_0	I_1	I_0	I_1	I_0	I_1
I_0	I_2	I_1	I_3		
I_1	I_2	I_2	I_3		
I_1	I_3				

2.2 Given four instructions, how many unique comparisons (between register sources and destinations) are necessary to find all of the RAW, WAR and WAW dependencies (find a tight upper bound). Answer for the case of four instructions, and then derive a general equation for N instructions. Assume that all instructions have one register destination and two register sources.

For RAW dependencies for instruction K:

$$\frac{1 \text{ comparison}}{\text{previous instruction} \times \text{src operand/instruction}} \times \frac{2 \text{ src operands}}{\text{instruction}} \times K \text{ previous instructions}$$

$= 2K$ comparisons

For WAR ~~instruction~~ dependencies for instruction K:

$$\frac{2 \text{ comparisons}}{\text{previous instruction} \times \frac{\text{dest operands}}{\text{instruction}}} \times \frac{1 \text{ dest operand}}{\text{instruction}} \times K \text{ previous instructions}$$

$= 2K$ comparisons

For WAW dependencies for instruction K:

$$\frac{1 \text{ comparison}}{\text{prev. instr.} \times \frac{\text{dest operand}}{\text{instr}}} \cdot \frac{1 \text{ dest operand}}{\text{instr}} \cdot K \text{ prev. instr} = K \text{ comps.}$$

$$\text{Total comparisons} = \sum_{K=0}^{N-1} (K + 2K + 2K) = \boxed{\frac{5(N)(N-1)}{2}}$$

When $N = 4$

$$\frac{5(4)(3)}{2} = 30 \text{ comps.}$$

3 Loop Unrolling and Fibonacci

Consider the following pseudo-C code to compute the fourth Fibonacci number ($F(4)$).

```
1  int a,b,i,t;
2  a=b=1;       /* Set a and b to F(2) and F(1) respectively */
3  for(i=0;i<2;i++)
4  {
5    t=a;       /* save F(n-1) to a temporary location */
6    a+=b;      /* F(n) = F(n-1) + F(n-2) */
7    b=t;       /* set b to F(n-1) */
8  }
```

3.1 Convert the pseudo-C code for this snippet of pseudo-C code into reasonably efficient MIPS assembly code. Represent each variable of the pseudo-C program with a register. Clearly indicate which register corresponds to which variable. Try to follow the pseudo-C code as closely as possible, but do not use any pseudo-instructions except for `mov`.

```
        addi  $t0, $zero, 1     ; a = 1
        addi  $t1, $zero, 1     ; b = 1
        add   $t2, $zero, $zero ; i = 0
        addi  $t4, $zero, 2     ; temp = 2
loop:   add   $t3, $zero, $t0   ; t = a
        add   $t0, $t0, $t1     ; a = a + b
        add   $t1, $zero, $t3   ; b = t
        add   $t2, $t2, 1       ; i = i + 1
        slt   $t5, $t2, $t4     ; $t5 = i < 2
        bneq  $t5, $zero, loop
```

One observation that a compiler might make is that the loop construction is somewhat unnecessary. Since the the range of the loop indices is fixed, one can *unroll* the loop by simply writing three iterations of the loop one after the other without the intervening increment/comparison on `i`. For example, the above could be written as:

```
1  int a,b,t;
2  a=b=1;
3  t=a;
4  a+=b;
5  b=t;
6  t=a;
7  a+=b;
8  b=t;
```

Convert the pseudo-C code for this snippet of pseudo-C code into reasonably efficient MIPS assembly code. Represent each variable of the pseudo-C program with a register. Clearly indicate which register corresponds to which variable. Try to follow the pseudo-C code as closely as possible, but do not use any pseudo-instructions except for `mov`.

```
addi  $t0, $zero, 1    ; a=1
addi  $t1, $zero, 1    ; b=1
add   $t2, $zero, $t0  ; t=a
add   $t0, $t0, $t1    ; a=a+b
add   $t1, $zero, $t2  ; b=t
add   $t2, $zero, $t0  ; t=a
add   $t0, $t0, $t1    ; a=a+b
add   $t1, $zero, $t2  ; b=t
```

3.2 Now suppose that instead of the fourth Fibonacci number we decided to compute the 20th (18 iterations). How many static instructions would there be in the first version and how many would there be in the unrolled version? What about dynamic instructions? *You do not need to write out the assembly for this part.*

"Rolled" Version

static 10 (changing # of iterations doesn't change the program)

dynamic 6 loop instrs · 18 iterations + 4 before loop = 112 instrs

Unrolled Version

3 instrs per "iteration" · 18 + 2 = 56

56, static == dynamic

3.3 Even if the non-unrolled and unrolled versions of code had the same dynamic instruction count, what would be the advantage of unrolled code on a deeply pipelined system?

There are no branches in the unrolled version which means no pipeline bubbles are introduced due to branch misprediction. On deep pipelines, branch mispredictions are expensive because of the time to detect and recover (flush the pipe).

3.4 Assuming a standard MIPS five-stage, single-issue pipe with no branch prediction how many cycles will the unrolled and non-unrolled versions of $F(20)$ take? *State any assumptions you make.*

Assumptions: All forwarding paths present
1 cycle taken branch stall

Unrolled version: 56 dynamic instrs · $\frac{1 \text{ cycle}}{\text{instr}}$ + 4 cycle pipeline fill =

60 cycles

"Rolled" version: 112 dynamic instrs · $\frac{1 \text{ cycle}}{\text{instr}}$ + $\frac{1 \text{ cycle}}{\text{branch}}$ × 18 branches

+ 4 cycle pipeline fill

= **134 cycles**

3.5 (Optional Extra Credit) If we only care about the value of a at the end of the program, how might a *really* smart compiler further optimize the unrolled code for $F(4)$?

The compiler could perform partial evaluation and realize that F(4) = 3. Therefore the code:

```
addi $t0, $zero, 3
```

is sufficient.

4 Pipelining and Bypass

In this question we will explore how bypassing affects program execution performance. To begin consider the standard MIPS 5 stage pipeline (presented in lecture and the book). For your reference, refer to the figure on the last page. For this question, we will use the following code to evaluate the pipeline's performance:

```
1   add  $t2, $s1, $sp
2   lw   $t1, $t2, 0
3   addi $t2, $t1, 7
4   add  $t1, $s2, $sp
5   lw   $t1, $t1, 0
6   addi $t1, $t1, 9
7   sub  $t1, $t1, $t2
```

4.1 What is the load-use latency for the standard MIPS 5-stage pipeline?

The load-use latency is 2 cycles. (1 bubble between load and use)

4.2 Once again, using the standard MIPS pipeline, identify whether the value for each register operand is coming from the bypass or from the register file. Recall that in the MIPS pipeline, writes occur in the first half of each cycle, while reads occur in the second half of each cycle. For clarity, please write REG or BYPASS in each box.

Instruction	Src Operand 1	Src Operand 2
1	REG	REG
2	BYPASS	N/A
3	BYPASS	N/A
4	REG	REG
5	BYPASS	N/A
6	BYPASS	N/A
7	BYPASS	REG

4.3 How many cycles will the program take to execute on the standard MIPS pipeline?

cycles = 7 cycles + 2 cycles + 4 cycles = 13cycles

7 cycles: # dynamic instructions

2 cycles: 2 load use bubbles

4 cycles: pipeline fill time

4.4 Assume, due to circuit constraints, that the bypass wire from the memory stage back to the execute stage is omitted from the pipeline. What is the load-use latency for this modified pipeline?

The load-use latency is 3 cycles (2 bubbles between load and use)

4.5 How long does the program take to execute on the modified pipeline?

cycles = 7 + 4 + 4 = 15 cycles

7: # dynamic instructions

4: 2 x 2 load-use bubbles

4: pipeline fill

4.6 Could the code be transformed to make the program perform better on the modified pipeline? If so, show the transformed program. You are free to use any register not used in the code. Also, you can assume that only register `$t1` is used by the code following this code. How many cycles does it take to execute on the modified pipeline? How about on the original pipeline?

```
1   add  $t2, $s1, $sp
2   lw   $t1, $t2, 0
4   add  $t11, $s2, $sp
5   lw   $t11, $t11, 0
3   addi $t2, $t1, 7
6   addi $t11, $t11, 9
7   sub  $t1, $t11, $t2
```

Fill the bubbles between load and use

Modified pipeline: 7 cycles + 1 cycle + 4 cycles = 12 cycles

7 cycles: dynamic instrs

1 cycle: load-use bubble between instrs 5 and 6

4 cycles: pipeline fill

Notice that counting the bubbles in this program is tricky. Because only 1 instruction lies between the load and the use in instructions 5 and 6, 1 bubble must be inserted after instruction 3 executes.

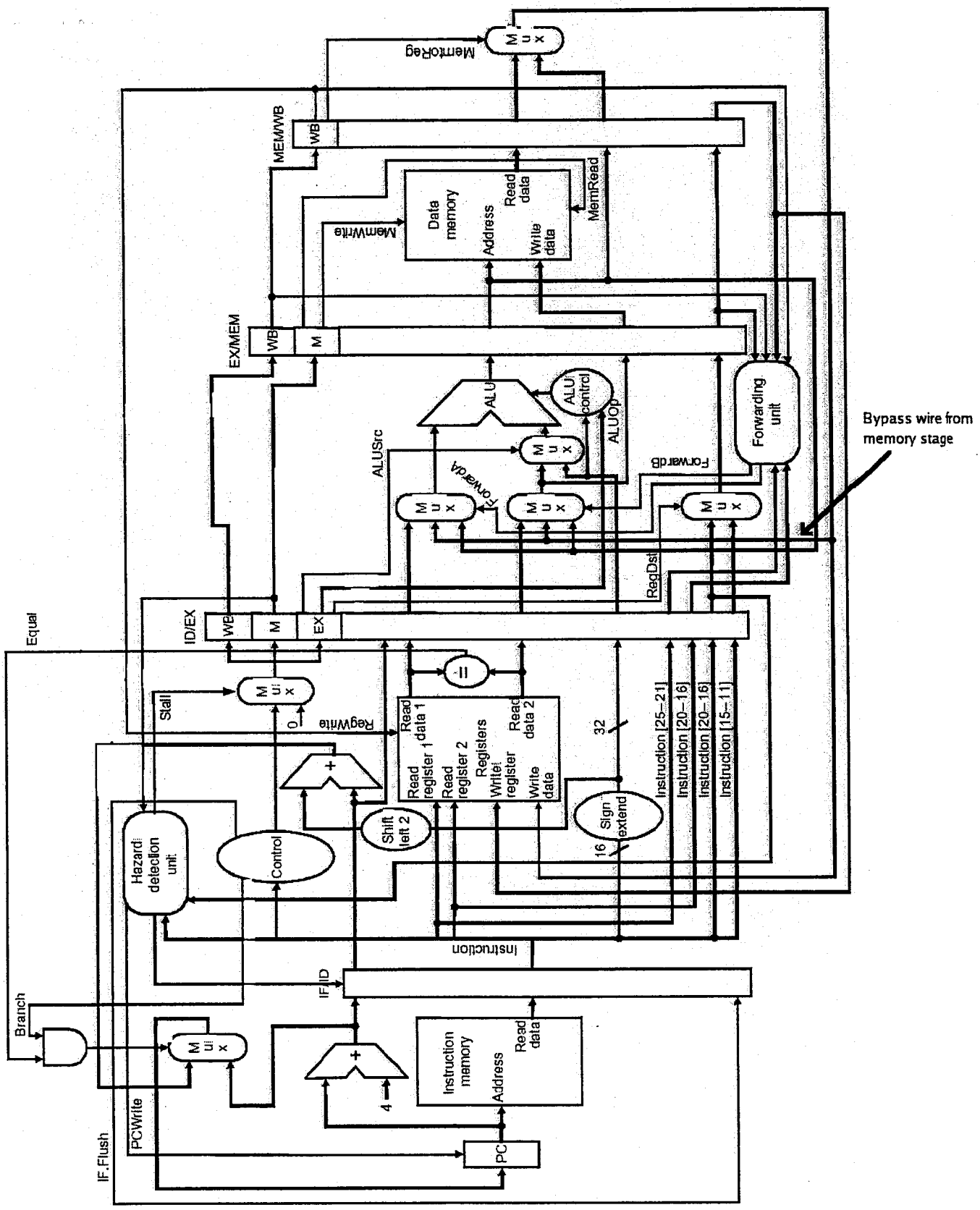
If the load-use did not cause a bubble, then the def-use (RAW) dependence between instructions 3 and 7 would cause a bubble (7 is in ID when 3 is in MEM & and no MEM to EX bypass exists. So 7 must wait in ID). However, this bubble does not occur because instruction 3 is already delayed by the load-use bubble from earlier.

original pipeline: 7 cycles + 0 cycles + 4 cycles = 11 cycles

↑ dynamic instrs

↑ load-use bubbles

↑ pipeline fill


MemtoReg
MEM/WB
WB
Data memory
Address
Read data
Write data
MemRead
MemWrite
EX/MEM
WB
M
ALU
ALU control
ALUOp
ALUSrc
ForwardA
ForwardB
Forwarding unit
Bypass wire from memory stage
RegDst
ID/EX
WB
M
EX
Equal
=
Stall
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
32
Instruction [25–21]
Instruction [20–16]
Instruction [20–16]
Instruction [15–11]
Sign extend
16
Shift left 2
Hazard detection unit
Control
Instruction
IF/ID
Branch
Instruction memory
Address
Read data
4
PCWrite
IF.Flush
PC
M
u
x
+