

Name: _____

Page 1 of 9

CS/ECE 552 Introduction to Computer Architecture

Midterm Exam

Tuesday, March 16, 1999

7:15-9:15 p.m

Name: _____ SOLUTION _____

Limit your answers to the space provided. Unnecessarily long answers will be penalized. If you use more space than is provided, you are probably doing something wrong. Use the back of each page for any scratch work. Write your last name on each page.

Problem 1. _____ (out of 24 points)

Problem 2. _____ (out of 14 points)

Problem 3. _____ (out of 8 points)

Problem 4. _____ (out of 12 points)

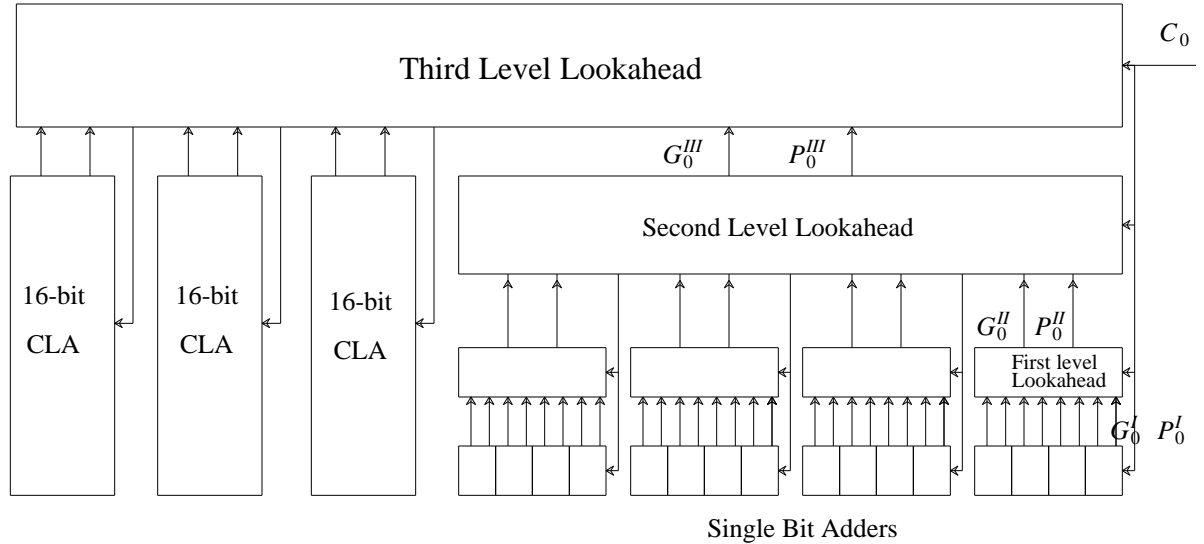
Problem 5. _____ (out of 18 points)

Problem 6. _____ (out of 24 points)

Total _____ (out of 100 points)

(1) Adder Design (24 points)

The figure below shows a 64-bit carry lookahead adder with three levels of lookahead. Some signals are shown in the figure to give you an idea of the structure of the adder; however not all relevant signals are shown (for example the carries from the first level lookahead to the single-bit adders).



- (a) (10 points) Let X_i, Y_i, C_i and S_i , with $i = 0, 1, \dots, 63$, represent the individual bits of the two operands, the carries, and the sums, respectively. Let G_i^I and P_i^I , $i = 0, \dots, 63$ represent the first level generates and propagates. Let G_j^II and P_j^II , $j = 0, \dots, 15$ represent the second level generates and propagates, and let G_k^III and P_k^III , $k = 0, \dots, 3$ represent the third level generates and propagates.

Write the logic equations for the following signals, in the space provided. (The equations should be in terms of the inputs to the logic block for which they represent the outputs.)

$G_3^I =$	$X_3 \cdot Y_3$
$G_2^{II} =$	$G_{11}^{II} + G_{10}^{II} \cdot P_{11}^{II} + G_9^{II} \cdot P_{11}^{II} \cdot P_{10}^{II} + G_8^{II} \cdot P_{11}^{II} \cdot P_{10}^{II} \cdot P_9^{II}$
$P_1^{III} =$	$P_7^{II} \cdot P_6^{II} \cdot P_5^{II} \cdot P_4^{II}$
$C_8 =$	$G_1^{II} + G_0^{II} \cdot P_1^{II} + P_0^{II} \cdot P_1^{II} \cdot C_0$
$C_{32} =$	$G_1^{III} + G_0^{III} \cdot P_1^{III} + P_0^{III} \cdot P_1^{III} \cdot C_0$

- (b) Assume that each gate delay is τ time units, and all gates are available with up to 5 inputs. Further assume that all X_i , all Y_i , and C_0 are ready at time 0. In the table below, show at what time the chosen signal value is ready. Use the comment column for any comments (comments will be used to determine partial credit in case of an incorrect answer). **(6 points)**

Signal	Time Ready	Comments
S_2	5τ	τ for G_i^I, P_i^I ; 2τ for C_2 , and 2τ for S_2
S_8	7τ	τ for G_i^I, P_i^I ; 2τ for G_i^{II}, P_i^{II} ; 2τ for C_8 , and 2τ for S_8
C_{32}	7τ	τ for G_i^I, P_i^I ; 2τ for G_i^{II}, P_i^{II} ; 2τ for G_i^{III}, P_i^{III} , and 2τ for C_{32}

- (c) Calculate the number of gates of each type in *the lookahead logic* (include all levels and do not count gates in the 64 single-bit adders). Show your work. **(8 points)**

There are a total of $16+4+1 = 21$ CLA units. Each CLA unit generates a G, a P, and 3 carries. The equations for this logic are:

$$G = g_3 + g_2.p_3 + g_1.p_3.p_2 + g_0.p_3.p_2.p_1 \quad (3 \text{ AND and } 1 \text{ OR gate})$$

$$P = p_3.p_2.p_1.p_0 \quad (1 \text{ AND gate})$$

$$c_3 = g_2 + g_1.p_2 + g_0.p_1.p_2 + p_0.p_1.p_2.c_0 \quad (3 \text{ AND gates and } 1 \text{ OR gate})$$

$$c_2 = g_1 + g_0.p_1 + p_0.p_1.c_0 \quad (2 \text{ AND and } 1 \text{ OR gate})$$

$$c_1 = g_0 + p_0.c_0 \quad (1 \text{ AND and } 1 \text{ OR gate})$$

So each unit has 10 AND gates and 4 OR gates (total of 14 gates). So the overall lookahead logic has $10 \times 21 = 210$ AND gates and $4 \times 21 = 84$ OR gates.

(2) **Short Questions (14 points)**

- (i) What are *delayed branches*? **(3 points)**

A branch instruction for which the instruction following the branch is unconditionally executed regardless of the outcome of the branch.

- (ii) What problem do *delayed branches* solve, and how? **(3 points)**

Delayed branches solve the problem of branch hazards. By unconditionally executing an instruction following a branch, the pipeline can be kept full, i.e., without any bubbles, until the outcome of the branch is known.

- (iii) The multicycle datapath that we studied had one ALU, which was used to increment the PC as well as for ALU operations (and address calculations), and branch target calculation. The single-cycle datapath had 3 separate adders/ALUs, one for incrementing the PC, one for ALU operations (and address calculation), and one for branch target calculation. Why are a different number of ALUs used in the two different datapaths? **(4 points)**

In the single-cycle datapath, we needed to perform 3 ALU operations in a clock cycle, and since a single ALU could only perform one operation in a cycle, 3 ALUs were needed. In the multicycle datapath, the same ALU could be used for all 3 functions, in different clock cycles.

- (iv) Is it easier to have a pipelined implementation for an architecture which has fixed-sized instructions (e.g., 4 bytes) than for an architecture which has variable-sized instructions? Why or why not? **(4 points)**

It is easier to have a pipelined implementation for an architecture with fixed-size instructions because there is no need to decode an instruction (to find out its length) in order to be able to determine where the next instruction starts. This allows IF and ID to be overlapped.

(3) Machine Performance (8 points)

Consider two different implementations M1 and M2 of the same instruction set. There are four classes of instructions (A, B, C, and D) in the instruction set.

M1 has a clock rate of 350 MHz and M2 has a clock rate of 450 MHz.

The fraction of all instructions that belong to a particular class, and the average number of cycles for each instruction in the two implementations are as below:

Class	Percentage of instructions in class	CPI on M1	CPI on M2
A	10%	1	2
B	35%	2	2
C	25%	3	4
D	30%	4	4

Which implementation of the instruction set is faster, and by how much? Show your work.

$$\text{Overall CPI on M1} = 0.1 \times 1 + 0.35 \times 2 + 0.25 \times 3 + 0.3 \times 4 = 2.75$$

$$\text{Overall CPI on M2} = 0.1 \times 2 + 0.35 \times 2 + 0.25 \times 4 + 0.3 \times 4 = 3.1$$

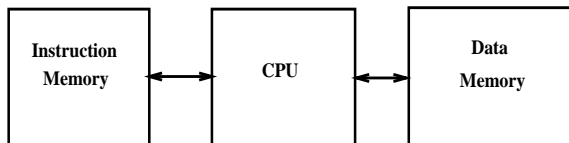
$$\text{Time on machine} = \text{CPI} \times \text{cycle time} \times \text{number of instructions.}$$

$$(\text{Time on M1}) / (\text{Time on M2}) = (2.75 / 350 \times 10^6) / (3.1 / 450 \times 10^6) = 1.14$$

So M2 is 14% faster than M1.

(4) **System Balance (12 points)**

Consider a system with a CPU, connected to separate instruction and data memories, as shown in the figure below.



The system is executing the following MIPS code. The code is a loop, which you can assume iterates many times. Also assume that the `beq` instruction is taken 25% of the time.

```

$L6:
    lw    $2,0($4)
    beq   $2,$0,$L4
    lw    $2,0($5)
    lw    $3,0($6)
    addu  $2,$2,$3
    sw    $2,0($4)

$L4:
    addu  $4,$4,4
    addu  $6,$6,4
    addu  $5,$5,4
    addu  $7,$7,1
    slt   $2,$7,10000
    bne   $2,$0,$L6
  
```

Remember that in MIPS the instruction size is 4 bytes, and the word size is also 4 bytes.

- (i) What bandwidth would the instruction memory need to have to be able to support a sustained instruction execution rate of 200 MIPS when executing the above program? Show your work. **(8 points)**

When the branch is not taken, there are 12 instructions executed and 4 memory references made per loop iteration. When the branch is taken, there are 8 instructions executed, and 1 memory reference made.

On average, there are $0.75 \times 12 + 0.25 \times 8 = 11$ instructions and $4 \times 0.75 + 1 \times 0.25 = 3.25$ memory references per iteration.

For each instruction we need to fetch 4 bytes, so to support a sustained instruction execution rate of 200 MIPS we need to have an instruction memory bandwidth of $200 \text{ MIPS} \times 4 \text{ bytes per instruction} = 800 \text{ Mbytes per second}$

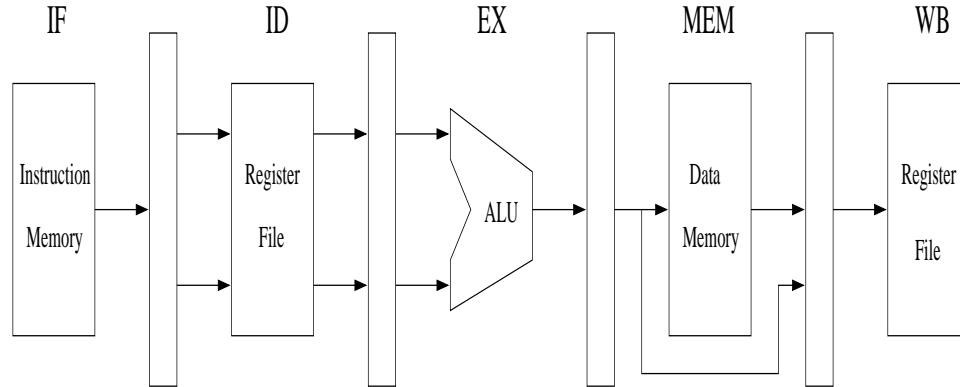
- (ii) What bandwidth would the data memory need to have to be able to support a sustained instruction execution rate of 200 MIPS when executing the above program? Show your work. **(4 points)**

As above, there are 3.25 memory references per 11 instructions on average. Each memory reference is 4 bytes. So the bandwidth required from the data memory to support an execution rate of 200 MIPS is

$$200 \times (3.25 / 11) \times 4 = 236.36 \text{ Mbytes per sec.}$$

(5) **Pipelining (18 points)**

The following figure shows a simple pipeline. This pipeline has a register file that WRITES during the FIRST half of the clock cycle and READS during the SECOND half of the clock cycle. Instructions can be stalled only in the instruction-fetch and instruction-decode stages.



(a) For the following instruction sequence, show the pipeline timing in the table below. Label pipeline bubbles caused by stalls with "bubble". (Note: You may not need all 16 cycles provided in the table.) **(8 points)**

```

load r2 <- mem(r1)    ; LOAD
r2 <- r1 + r2         ; ADD1
r3 <- r1 + r3         ; ADD2
store mem(r4) <- r2   ; STORE
    
```

Cycles	IF	ID	EX	MEM	WB
1	LOAD				
2	ADD1	LOAD			
3	ADD2	ADD1	LOAD		
4	ADD2	ADD1	bubble	LOAD	
5	ADD2	ADD1	bubble	bubble	LOAD
6	STORE	ADD2	ADD1	bubble	bubble
7		STORE	ADD2	ADD1	bubble
8		STORE	bubble	ADD2	ADD1
9			STORE	bubble	ADD2
10				STORE	bubble
11					STORE
12					
13					
14					
15					
16					

THIS QUESTION CONTINUES ON THE NEXT PAGE.

Pipelining question continued

- (b) Eliminate all of the bubbles caused by the instruction sequence in part (a) by reordering the instruction sequence and/or by adding bypass paths to the pipeline diagram on the previous page. Show the reordered instruction sequence in the space below, and any modifications to the pipeline diagram in the figure on the previous page. For your convenience, the original code sequence from the previous page is replicated below. Be sure to explain how your solution eliminates the bubbles that existed. **(10 points)**

```
load r2 <- mem(r1)    ; LOAD
r2 <- r1 + r2         ; ADD1
r3 <- r1 + r3         ; ADD2
store mem(r4) <- r2   ; STORE
```

The first bubble after the LOAD instruction cannot be removed using a bypass. In order to eliminate it the ADD1 and ADD2 instructions are reversed. Notice that they are independent of each other, so they can be moved in this way without affecting the semantics of the code segment. This creates an additional bubble before the STORE instruction, but it can be removed using bypasses.

The two original bubbles that remain are removed by adding a bypass from both of the inputs into the MEM/WB pipeline register to both of the inputs of the ID/EX pipeline register. The new bubble that was introduced by the code motion is removed by adding a bypass from the input of the EX/MEM pipeline register to both of the inputs of the ID/EX pipeline register.

(6) Datapath (24 points)

In this problem you will extend the multi-cycle datapath as presented in the course text in Figures 5.33 and 5.34. These figures have been provided on the last page of this exam.

It is possible to decrease the execution time of a program by combining two or more instructions (that would normally take two or more instruction times to execute) into a single instruction that takes only one instruction time (or a fraction more than one) to execute. These hybrid instructions are chosen because the instructions they replace appear often as a group in programs, and because they can be implemented without extensive modification to the datapath.

Modify the datapath as necessary to allow the implementation of each instruction. Be sure to include complete information about any new signals that appear in the modified datapath.

After the datapath is modified, fill in the list of control signals that must appear at each clock cycle in order to implement the instruction. To reduce the number of signals that you have to write, assume that the initial value of all signals is zero. Remember that if you assert a signal, you must de-assert it when necessary. The number of clock cycles needed for each instruction may vary. Use as many as you need, but make the implementation as efficient as possible. The signals for the first clock cycle already appear in the table. You should not have to modify them.

ALL OF YOUR WORK FOR EACH INSTRUCTION SHOULD APPEAR ONLY ON THE PAGE FOR THAT INSTRUCTION.

Note: In figure 5.34, ALU stands for arithmetic logic unit, MDR for memory data register, PC for program counter, and IR for instruction register.

In this space was Figure 5.33 from the course text.

Circuitry added to the datapath:

The MUX on the Write register input to the register file is expanded to allow input 2 to be bits [25-21] of the instruction register. This allows Rs to be written.

The MUX on the B ALU input is expanded to allow input 4 to be a hard-wired "1".

Control:

Note: Signals in parens are not needed (but should be included for clarity) because they were already set to that value. This includes those assumed to be zero at the start of the instruction.

- (a) The DBNZ Rs,offset instruction decrements the value in register Rs (be sure to write it back to Rs) and branches to the target computed by adding the signed offset in bits [15-0] of the instruction to the PC, if the result of the decrement is non-zero. (**Note:** you may not need all the clock cycles provided in the table below.)

Clock Cycle	Functional Description	Signals
1	Instruction -> IR and PC=PC+4	MemRd=1, ALUSrcA=0, IorD=0, IRWrite=1, ALUSrcB=01 ALUOp=00, PCWrite=1, PCSource=00
2	Read Rs and Rt and compute branch target	MemRd=0, IRWrite=0, PCWrite=0 (ALUSrcA=0), ALUSrcB=11, (ALUOp=00)
3	Increment Rs and Update the PC	ALUSrcA=1, ALUSrcB=100, (ALUOp=00) PCSource=01, PCWriteCond=1
4	Write back Rs	RegDst=10, RegWrite=1, PCWriteCond=0
5	Finish	RegWrite=0
6		
7		
8		

In this space was Figure 5.33 from the course text.

Circuitry added to the datapath:

The MUX on the Write register input to the register file is expanded to allow the 2 to be bits [25-21] of the instruction register. This allows Rs to be written.

The MUX on the A ALU input is expanded to allow input 2 to be the output of the register file B output holding register.

Control:

Note: Signals in parens are not needed (but should be included for clarity) because they were already set to that value. This includes those assumed to be zero at the start of the instruction.

- (b) The LWI Rs,immediate(Rt) instruction loads the word addressed by the sum of the sign-extended immediate and the value in Rt into register Rs. The value in register Rt is also auto-incremented so that the next execution of the instruction will load the next **word** in memory. (**Note:** you may not need all the clock cycles provided in the table below.)

Clock Cycle	Functional Description	Signals
1	Instruction -> IR and PC=PC+4	MemRd=1, ALUSrcA=0, IorD=0, IRWrite=1, ALUSrcB=01 ALUOp=00, PCWrite=1, PCSource=00
2	Read Rt and compute branch target	MemRd=0, IRWrite=0, PCWrite=0 (ALUSrcA=0), ALUSrcB=11, (ALUOp=00)
3	Compute the load address	ALUSrcA=10, (ALUSrcB=11), (ALUOp=00)
4	Get the value and Compute the new Rt	MemRd=1, IorD=1 (ALUSrcA=10), ALUSrcB=01, (ALUOp=00)
5	Write the new Rt	MemRd=0, (MemtoReg=0), (RegDst=0), RegWrite=1
6	Write the new Rs	MemtoReg=1, RegDst=10, (RegWrite=1)
7	Finish	RegWrite=0
8		