Name **Midterm SOLUTIONS**

CS 561 A97 Midterm Examination

November 5, 1997

Instructions: This exam is open book and open notes. There are 27 pages - don't panic most of it is blank workspace. Put your name on top of each one NOW! The exam is worth 100 points - each point is worth a minute if you hope to get a perfect score - so plan your time. The intent is that the difficulty increases with the number of the problem but you should look at them all and plan your strategy. The last problem is definitely the hardest - a.k.a. the *thought* problem.

There should be ample workspace provided - but feel free to use additional scratch paper if you need it - however make sure that you show your work in places where we can find it - also make sure your name is on each page and that things are collated properly when you turn it in - there will be a stapler available. Write legibly - you'll lose points if we can't read what you write.

TABLE 1.

Problem	**Subproblem**	**Points**	**Grader**	**Score**
1		15	Al	
2		10	Al	
	a	5		
	b	5		
3		20	Uros	
4		30	Uros	
5	*THOUGHT*	25	Al	
	a	2		
	b	3		
	c	2		
	d	15		
	e	3		
TOTAL		**100**		

Name **Midterm SOLUTIONS**

Problem 1: (15 points)

Given the following instruction distributions and cycle times for a perfect memory system machine (from page 45, Fig. 1.17 in the text):

Instruction Type	Frequency	Clock cycles
ALU ops	43%	1
Loads	21%	2
Stores	12%	2
Branches	24%	2

A perfect memory system implies that every data and instruction access hits in the cache. However measurements show that instructions accesses miss in the cache 5% of the time and data references miss 10% of the time. When either type of miss is encountered an extra 40 cycles is required to complete the instruction. This is often called the miss penalty.

How much faster does the machine run with the ideal memory than with the real memory system?

(Hint: useful equations can be found on pages 32, 33, 34, and 43 of the text)

Since the instruction count and cycle time does not vary whether we have an ideal or a real memory system - calculating the slow down will simply be the ratio of CPIideal/CPIreal. Every instruction must be fetched in order to execute but only loads and stores will result in data accesses. Hence the table can be expanded to be:

Inst. Type	Frequency	Clock Cycles	Inst. Accesses	Data Accesses
ALU ops	43%	1	1	0
Loads	21%	2	1	1
Stores	12%	2	1	1
Branches	24%	2	1	0

CPIreal = CPIideal + CPId-stalls + CPIi-stalls (where i=instruction and d=data).

CPIi-stalls = 5% x 40 = 2 cycles

CPId-stalls = 10% x 40 = 4 cycles

by adding these stall terms in cycles into the appropriate column of the above table we can now derive a table that shows the CPIideal and CPIreal figures:

Instruction Class	**Frequency**	**CPIideal**	**CPIreal**
ALUops	43%	1	3
Loads	21%	2	8
Stores	12%	2	8
Branch	24%	2	4

Now we can compute aggregate CPIideal and CPIreal by the formula on page 34 and by noting that ICi/instruction count is just the instruction frequency. Hence

$$\mathrm{CPI} = \sum_{\forall \mathrm{classes}} \mathrm{CPI}_{\mathrm{class}} \times \mathrm{Frequency}_{\mathrm{class}}$$

Hence CPIideal = (1 x .43) + (2 x .21) + (2 x .12) + (2 x .24) = 1.57

and CPIreal = (3 x .43) + (8 x .21) + (8 x .12) + (4 x .24) = 4.89

and finally the speedup = 4.89/1.57 = 3.11465

Hence the ideal memory system performance is 3.11 times faster than the performance of the machine with a real memory system.

Problem 2: (10 points)

We are considering adding a new "displaced indexed" address mode to the DLX. This will allow code sequences of the form:

```
add    r1, r1, r2
lw     r4, 0(r1)
```

to be replaced by single instruction using the new address mode and a shorter 11 bit offset:

```
lw     r4, 0(r1, r2)
```

Using the overall average instruction frequencies shown in Figure 2.26 on page 105 of your text - evaluate this option via the following 2 questions.

Part a: (5 points)

Assuming the new addressing mode can be used for 10% of the displacement loads and stores (accounting for both the frequency of this type of address calculation and the shorter offset)

What is the ratio of the instruction count on the enhanced DLX compared to the old DLX?

From Figure 2.26 we find:

loads are 26% of the mix
stores are 9% of the mix
adds are 14% of the mix

The key here is that for the 10% of the time that this enhancement can work for displaced loads and stores we get to remove the add that was really used to create the indexed part of the address calculation. Such adds constitute 3.5% of the instruction mix [(10% x 26%) + (10% x 9%)]. Let's call the old DLX the *classic* and the new DLX the *enhanced* versions.

Hence ICenhanced = .965 x ICclassic

Name **Midterm SOLUTIONS**

Part b: (5 points)

However the extra 2 stage addition process that is required to support the displaced indexed address mode lengthens the cycle time by 5%,

Which machine is faster and by how much?

This one is a simple calculation of our basic performance equation:

Speedup = CPU-TIMEclassic / CPU-TIMEenhanced

CPU-TIME = CPI x Instruction-count x cycle-time

Hence

Speedup = CPIclassic x ICclassic x CTclassic / CPIclassic x .965 ICclassic x 1.05 CTclassic)

= 0.987

Which implies that the classic DLX is actually 1.3% faster than the enhanced version and therefore the enhancement should not be made.

[Note - this is a classic variation on the Amdahl Law theme - namely any enhancement that only applies 3.5% of the time (as in this case) better be free and not have a 5% penalty associated with it as in this instance.]

Name **Midterm SOLUTIONS**

Problem 3: (20 points)

Create a table for the R4000 integer pipeline shown in Figure 3.50 (page 201) that is similar to the DLX table shown in Figure 3.19 (page 160) Consider only the forwarding for register to register ALU, register - immediate ALU, and Load instructions. You DO NOT have to do the comparison column.

Note the following:

Values coming from ALU instructions can be forwarded from any of the stages from EX/DF, DF/DS, DS/TC, or TC/WB as needed by later instructions

Loads can come from both the DS/TC and TC/WB stages in the R4000 pipe since data is available at the end of DS even though the result of the tag match has not been reported.

The term ANY means any of the instruction types

Remember forwarding is required if the destination of an earlier instruction is the source for a subsequent instruction.

The table is shown on the next page.

Blank Workspace Page

Source Pipeline Register	Source Opcode	Destination Pipeline Register	Destination Opcode	Destination of Result
EX/DF	Reg-Reg ALU	RF/EX	Any	ALU Top
EX/DF	Reg-Reg ALU	RF/EX	Reg-Reg ALU	ALU Bottom
DF/DS	Reg-Reg ALU	RF/EX	Any	ALU Top
DF/DS	Reg-Reg ALU	RF/EX	Reg-Reg ALU	ALU Bottom
DS/TC	Reg-Reg ALU	RF/EX	Any	ALU Top
DS/TC	Reg-Reg ALU	RF/EX	Reg-Reg ALU	ALU Bottom
TC/WB	Reg-Reg ALU	RF/EX	Any	ALU Top
TC/WB	Reg-Reg ALU	RF/EX	Reg-Reg ALU	ALU Bottom
EX/DF	ALU-Immed	RF/EX	Any	ALU Top
EX/DF	ALU-Immed	RF/EX	Reg-Reg ALU	ALU Bottom
DF/DS	ALU-Immed	RF/EX	Any	ALU Top
DF/DS	ALU-Immed	RF/EX	Reg-Reg ALU	ALU Bottom
DS/TC	ALU-Immed	RF/EX	Any	ALU Top
DS/TC	ALU-Immed	RF/EX	Reg-Reg ALU	ALU Bottom
TC/WB	ALU-Immed	RF/EX	Any	ALU Top
TC/WB	ALU-Immed	RF/EX	Reg-Reg ALU	ALU Bottom
DS/TC	Load	RF/EX	Any	ALU Top
DS/TC	Load	RF/EX	Reg-Reg ALU	ALU Bottom
TC/WB	Load	RF/EX	Any	ALU Top
TC/WB	Load	RF/EX	Reg-Reg ALU	ALU Bottom

Problem 4: (30 points)

This problem involves mentally simulating the execution of a program using scoreboarding for a 3-issue (3 way superscalar) machine that is a bit more real than the ones you have worked with so far.

Given the following twice unrolled unscheduled version of the SAXPY loop:

```
LD      F2, 0(R1)
MULTD   F4, F2, F0
LD      F6, 0(R2)
ADDD    F6, F4, F6
SD      0(R2),F6
ADDI    R1, 8
ADDI    R2, 8
LD      F2, 0(R1)
MULTD   F4, F2, F0
LD      F6, 0(R2)
ADDD    F6, F4, F6
SD      0(R2), F6
ADDI    R1, 8
ADDI    R2, 8
```

We will now see how well an out of order, multi-issue scoreboard does. Assume a perfect memory system (namely all LD and SD instructions hit in the cache). Assume you have an architecture with two integer Units (IUM and IU). Both can do integer operations in a single cycle. IUM can also do effective address calculation and access the cache in a single cycle. There are also two floating point units (FUA and FUM). FUA can perform ADDD instructions in 2 cycles, and FUM can perform MULTD instructions in 3 cycles.

The pipeline structure given the above function units is the familiar: issue, read operands, execute (using IUM, IU, FUA, and FUM), followed by write results. However both integer units are fully pipelined and the FUA and FUM units are not pipelined. The instruction window contains 8 slots and instructions can issue out of order but if multiple issues are possible then the choice is controlled by program order. In any cycle at most 2 integer instruction and 1 floating point instruction can issue. Similarly at most 2 integer instruction and 1 floating point instruction can

write their results, and at most 2 instructions can enter the instruction window per cycle. In order to maintain program order in the instruction window for the purpose of handling precise exceptions - instructions leave the window in program order. Note however that we won't otherwise deal with exceptions. LD and SD instructions must be issued to the IUM, other integer instructions are issued to IU if the IU is available, otherwise they will be issued to IUM. Since both IU and IUM are pipelined they can each be issued 1 new instruction per cycle. The functional units are considered busy until they finish the execution stage of the pipe then they are freed. This implies renaming register capability in the scoreboard that can contain values pending clearing the write result stage. To simplify assume that you always have enough.

Forwarding of the pipelined IU and IUM results is handled by a subscript in the following tables which in reality is just a generation tag that gets assigned by the scoreboard logic.

The additional rules are:

Issue: If an execution unit is available, and no other previous instruction has the same destination register, and the instruction does not depend on a result from any non-issued instruction needing the same functional unit, then it can issue. (These rules eliminate WAW and structural hazards as well as guaranteeing that deadlock can not occur.) At most 2 integer and 1 floating point instruction can issue on a single cycle. For issue, LD and SD are considered integer instructions. Given the renaming register capability the WAW dependency ends when the execution phase completes.

Read operands: completes in 1 cycle if the operands are in the register file. If not, then the most recent producer of the needed operand must finish it's execution stage and read operands can be completed in the same cycle. Note this implies a fully forwarded machine. For example - if some ADDD instruction is stalled in read results awaiting completion of a MULTD instruction that will finish it's 3 cycle execution phase after clock 5 then the read operands stage of the ADDD pipeline also completes after clock 5. (This step effectively resolves RAW hazards dynamically.) At most 2 integer and 1 floating point instruction can clear read operands in a given cycle.

Execution: execution cycles have specified. FUA and FUM are considered busy from the time they issue to the time they finish execution but are free during the write

results stage. IU and IUM are fully pipelined and can take a new instruction every cycle.

Write result: completes in 1 cycle if there is no WAR dependency. If there is a WAR dependency then this stage will complete in the same cycle that the antidependency clears it's read operands stage. At most 2 integer and 1 floating point instruction can clear write result in a single cycle. For write result, LD is considered a floating point instruction since the destination register is a floating point register.

Instruction window retire: At most 3 instructions can be retired (e.g. removed) from the instruction window in a given cycle and they must be removed in program order. If the top element of the instruction window has finished write result then it is retired from the window in the same cycle. The same is true for the second and third element in the window that are either writing results on the same cycle or have already finished writing results.

Assume that after clock 0 the instruction window is already full but nothing else has happened - then complete the tables for the next 10 clock cycles given the stated rules.

In order to make your answers more consistent and ease grading confusion use the following formats. When a stage is completed mark an X (feel free to use a numeric subscript for cycle accounting if you like). Since execution takes a number of cycles - enter a number in this column of the table indicating which stage of execution has just finished - e.g. for FUM you would enter 1, then 2, then X_3 for the execution phase of the pipe. Since all SD and LD use displaced addresses consider the address operand as both the register and the immediate value, note that SD also requires a write value which you should consider the right operand. Put immediate values in the Fj and Fk fields. Under Rj and Rk use the following convention: Y means the operand is available, N means not, and blank means either the instruction has already cleared read operands or has not yet been issued.

The other wrinkle will be function unit status IUM1, IUM2 and IU1 and IU2. Since these units are fully pipelined then these represent ordering of instructions in read operands or execution stages - they are treated like a queue where IUM1 is the front and IUM2 is the rear (following instruction). Also treat the instruction window as a queue with the oldest instruction at the top.

Now fill out the tables: 6 cycles gets you full credit, 4 more gets you extra!

Name **Midterm SOLUTIONS**

After Clock 1:

Instruction Window/Instruction Status

Instruction	Issue	Read Operands	Execution	Write Result
LD F2, 0(R1)	X			
MULTD F4, F2, F0	X			
LD F6, 0(R2)	struc			
ADDD F6,F4,F6	waw			
SD 0(R2), F6	struc			
ADDI R1,8	X			
ADDI R2,8	struc			
LD F2, 0(R1)	waw & struc			

Functional Unit Status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
IUM1	Y	LD	F2	0(R1)				Y	
IUM2	N								
IU1	Y	ADDI	R1	R1	8			Y	Y
IU2	N								
FUA	N								
FUM	Y	MULT	F4		F0	IUM1		N	Y

FP & Integer Register Result Status

	R1	R2	F0	F2	F4	F6
FU	IU1			IUM1	FUM	

Note - for clarity I have indicated the type of stall preventing progress - it's fine if you just left these entries blank. In this cycle we issued the full 2 int and 1 float instructions. Also the way I'm treating the pipelined IU's here is somewhat bogus since as described the xx1 and xx2 are really tags - however when the front one stalls the next one will stall too - so it's not that bogus.

After Clock 2:

Instruction Window/Instruction Status

Instruction	Issue	Read Operands	Execution	Write Result
LD F2, 0(R1)	X	X		
MULTD F4, F2, F0	X	raw		
LD F6, 0(R2)	X			
ADDD F6,F4,F6	waw			
SD 0(R2), F6	struc			
ADDI R1,8	X	X		
ADDI R2,8	X			
LD F2, 0(R1)	waw & struc			

Functional Unit Status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
IUM1	Y	LD	F2	0(R1)					
IUM2	Y	LD	F6	0(R2)				Y	
IU1	Y	ADDI	R1	R1	8				
IU2	Y	ADDI	R2	R2	8			Y	Y
FUA	N								
FUM	Y	MULT	F4		F0	IUM1		N	Y

FP & Integer Register Result Status

	R1	R2	F0	F2	F4	F6
FU	IU1	IU2		IUM1	FUM	IUM2

This cycle: LD F6 issues - SD can't issue since it goes to IUM and LD already did that this cycle, ADDI R2 issues. The other two instructions stall due to WAW, and in the second LD's case there is a need for IUM as well.

After Clock 3:

Instruction Window/Instruction Status

Instruction	Issue	Read Operands	Execution	Write Result
LD F2, 0(R1)	X	X	X	
MULTD F4, F2, F0	X	X		
LD F6, 0(R2)	X	X		
ADDD F6,F4,F6	waw			
SD 0(R2), F6	X			
ADDI R1,8	X	X	X	
ADDI R2,8	X	X		
LD F2, 0(R1)	struc			

Functional Unit Status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
IUM1	Y	LD	F6	0(R2)					
IUM2	Y	SD	Mem	0(R2)	F6			Y	N
IU1	Y	ADDI	R2	R2	8				
IU2	N								
FUA	N								
FUM	Y	MULT	F4		F0	IUM1			

FP & Integer Register Result Status

	R1	R2	F0	F2	F4	F6
FU		IU1			FUM	IUM1

This cycle: The SD can issue, the ADDD can't since it's destination register is also F6, the final LD no longer has a WAW dependency since the first LD F2 has now completed execution but it's also a memory instruction and can't issue in the same cycle as the SD.

After Clock 4:

Instruction Window/Instruction Status

Instruction	Issue	Read Operands	Execution	Write Result
MULTD F4, F2, F0	X	X	1	
LD F6, 0(R2)	X	X	X	
ADDD F6,F4,F6	X			
SD 0(R2), F6	X	raw		
ADDI R1,8	X	X	X	X
ADDI R2,8	X	X	X	
LD F2, 0(R1)	X			
MULTD F4, F2, F0	waw & struc			

Functional Unit Status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
IUM1	Y	SD	Mem	0(R2)	F6			Y	N
IUM2	Y	LD	F2	0(R1)				Y	
IU1	N								
IU2	N								
FUA	Y	ADDD	F6		F6	FUM		Y	N
FUM	Y	MULT	F4		F0	IUM1			

FP & Integer Register Result Status

	R1	R2	F0	F2	F4	F6
FU				IUM2	FUM	FUA

This cycle: LD F6 clears execution so ADDD can issue, SD can't complete read operands since it's RAW dependent on the ADDD, the second MULTD F4 enters the window since the first LD F2 finishes write result and vacates the window since it is the top instruction but the MULTD can issue since ADDD issued in this cycle and since there is a WAW dependence with the first MULTD that still hasn't completed execution. The second LD F2 issues.

After Clock 5:

Instruction Window/Instruction Status

Instruction	Issue	Read Operands	Execution	Write Result
MULTD F4, F2, F0	X	X	2	
LD F6, 0(R2)	X	X	X	X
ADDD F6,F4,F6	X	raw		
SD 0(R2), F6	X	raw		
ADDI R1,8	X	X	X	X
ADDI R2,8	X	X	X	war
LD F2, 0(R1)	X	struc		
MULTD F4, F2, F0	waw			

Functional Unit Status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
IUM1	Y	SD	Mem	0(R2)	F6			Y	N
IUM2	Y	LD	F2	0(R1)				Y	
IU1	N								
IU2	N								
FUA	Y	ADDD	F6		F6	FUM		Y	N
FUM	Y	MULT	F4		F0	IUM1			

FP & Integer Register Result Status

	R1	R2	F0	F2	F4	F6
FU				IUM2	FUM	FUA

This cycle: ADDD can’t finish read operands since it’s RAW dependent on the multiply which isn’t done yet, SD is at the front of the IUM pipe but can’t progress since it is RAW dependent on the ADDD, this in turn stalls LD F2 since this LD is behind the LD in the IUM pipe. The final MULTD can’t do anything since it’s WAW dependent on the still executing MULTD. The 2nd ADDI can’t write results because of a WAR dependence on SD 0(R2). Hence nothing really changed except MULTD, LD F6, and ADDI F2 move on 1 more click normally.

After Clock 6:

Instruction Window/Instruction Status

Instruction	Issue	Read Operands	Execution	Write Result
MULTD F4, F2, F0	X	X	X3	
LD F6, 0(R2)	X	X	X	X
ADDD F6,F4,F6	X	X		
SD 0(R2), F6	X	raw		
ADDI R1,8	X	X	X	X
ADDI R2,8	X	X	X	war
LD F2, 0(R1)	X	struc		
MULTD F4, F2, F0	X			

Functional Unit Status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
IUM1	Y	SD	Mem	0(R2)	F6			Y	N
IUM2	Y	LD	F2	0(R1)				Y	
IU1	N								
IU2	N								
FUA	Y	ADDD	F6		F6	FUM			
FUM	Y	MULT	F4		F0	IUM2		N	Y

FP & Integer Register Result Status

	R1	R2	F0	F2	F4	F6
FU				IUM2	FUM	FUA

This cycle: Nothing happens in the IUM pipe since SD is still stalled on the ADDD. But the first MULTD finally finishes execution and this allows the dependent ADDD to clear read operands, and the second MULTD to issue. The same WAR dependence persists.

After Clock 7:

Instruction Window/Instruction Status

Instruction	Issue	Read Operands	Execution	Write Result
ADDD F6,F4,F6	X	X	1	
SD 0(R2), F6	X	raw		
ADDI R1,8	X	X	X	X
ADDI R2,8	X	X	X	war
LD F2, 0(R1)	X	struc		
MULTD F4, F2, F0	X	raw		
LD F6, 0(R2)	waw & struc			
ADDD F6, F4, F6	waw & struc			

Functional Unit Status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
IUM1	Y	SD	Mem	0(R2)	F6			Y	N
IUM2	Y	LD	F2	0(R1)				Y	
IU1	N								
IU2	N								
FUA	Y	ADDD	F6		F6	FUM			
FUM	Y	MULT	F4		F0	IUM2		N	Y

FP & Integer Register Result Status

	R1	R2	F0	F2	F4	F6
FU				IUM2	FUM	FUA

This cycle: Finally something a bit interesting - MULTD finishes and hence both it and the following LD F6 are retired from the window and the next 2 instructions enter. However the IUM pipe is still stalled waiting for the add to finish and since the 2 new arrivals both have F6 as a destination register, this implies that they can't issue. The second MULTD needs F2 from the stalled LD. The WAR dependence persists.

After Clock 8:

Instruction Window/Instruction Status

Instruction	Issue	Read Operands	Execution	Write Result
ADDD F6,F4,F6	X	X	X2	
SD 0(R2), F6	X	X		
ADDI R1,8	X	X	X	X
ADDI R2,8	X	X	X	X
LD F2, 0(R1)	X	struc		
MULTD F4, F2, F0	X	raw		
LD F6, 0(R2)	struc			
ADDD F6, F4, F6	waw			

Functional Unit Status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
IUM1	Y	SD	Mem	0(R2)	F6				
IUM2	Y	LD	F2	0(R1)				Y	
IU1	N								
IU2	N								
FUA	N								
FUM	Y	MULT	F4		F0	IUM2		N	Y

FP & Integer Register Result Status

	R1	R2	F0	F2	F4	F6
FU				IUM2	FUM	

This cycle: Now the ADDD completes and SD can complete read operands. Sadly this is also where the IUM pipe specification of the problem is open to some interpretation - here I'm assuming that both parts of the pipe can't clear the same stage in the same cycle so while SD moves ahead, the LD F2 is stalled to finish this stage on the next cycle. If this is the case then the last LD F6 can't issue since the IUM pipe is still full. The ADDD can't issue because of a WAW dependency on a previous stalled instruction. The WAR dependency is now clear.

After Clock 9:

Instruction Window/Instruction Status

Instruction	Issue	Read Operands	Execution	Write Result
SD 0(R2), F6	X	X	X	
ADDI R1,8	X	X	X	X
ADDI R2,8	X	X	X	X
LD F2, 0(R1)	X	X		
MULTD F4, F2, F0	X	raw		
LD F6, 0(R2)	X			
ADDD F6, F4, F6	waw			
SD 0(R2), F6	struc			

Functional Unit Status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
IUM1	Y	LD	F2	0(R1)					
IUM2	Y	LD	F6	0(R2)				Y	
IU1	N								
IU2	N								
FUA	N								
FUM	Y	MULT	F4		F0	IUM1		N	Y

FP & Integer Register Result Status

	R1	R2	F0	F2	F4	F6
FU				IUM1	FUM	IUM2

This cycle: ADDD completes and retires, SD finishes execution, finally unplugging the IUM pipe, allowing LD F2 to advance past read operands, and LD F6 to issue. The second ADDD is still stuck with it's WAW dependence on LD F6. The newly arrived SD stalls since LD F6 issued and the IUM pipe is full.

After Clock 10:

Instruction Window/Instruction Status

Instruction	Issue	Read Operands	Execution	Write Result
ADDI R2,8	X	X	X	X
LD F2, 0(R1)	X	X	X	
MULTD F4, F2, F0	X	X		
LD F6, 0(R2)	X	X		
ADDD F6, F4, F6	waw			
SD 0(R2), F6	X			
ADDI R1, 8	X			
ADDI R2,8	struc			

Functional Unit Status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
IUM1	Y	LD	F6	0(R2)					
IUM2	Y	SD	Mem	0(R2)	F6			Y	N
IU1	Y	ADDI	R1	R1	8			Y	Y
IU2	N								
FUA	N								
FUM	Y	MULT	F4		F0	IUM1			

FP & Integer Register Result Status

	R1	R2	F0	F2	F4	F6
FU	IU1				FUM	IUM1

This cycle: Now the first SD finally completes but only 2 int's can retire so the second ADDI is stuck since only the first ADDI can retire. The ADDD can't issue since there is a WAW hazard with LD F6 that hasn't finished execution. But the final SD can issue even though it won't get it's F6 operand for a very long time as you saw in the first SD's case.

OK - this problem is too long but hopefully you learned something if you stuck with it.

Yikes - this problem was too time consuming.

Name **Midterm SOLUTIONS**

Problem 5: (25 points)

You've been given the thrilling job of optimizing a scientific application on a new RISC microprocessor architecture. This architecture is fully pipelined (including the FPU), but hazards can still arise from data-dependencies with penalties as given below :

Instruction Producing Result	Instruction Using Result	Latency (stalls)
FPU ALU Op	FPU ALU Op	3 cycles
FPU ALU Op	store	2 cycles
load	FPU ALU or INT ALU	1 cycle
FPU Compare	BFP{T,F}	1 cycle
FPU ALU Op	FPU Compare	3 cycles

Both conditional and unconditional branches (including jumps and calls) are handled by using a single-cycle delayed branch. A majority of the application's execution time is spent executing the following code fragment which takes two lists of data-points and calls some function foo() for all pairs of points that are closer than some specified distance max.

```
for (i = 0; i < n1; i++)
    for (j = 0; j < n2; j++) {
        dx = x[j] - x[i];
        dy = y[j] - y[i];
        r2 = dx*dx + dy*dy;
        if (r2 < max)
          foo();
    }
; R2 = &x[0]  These have been preloaded
; R3 = &y[0]
; R4 = n1
; R5 = n2
; F30 = max
```

The machine code is:

```
        ADD R6, R2, R0      ; R6 = &x[0]
        ADD R7, R3, R0      ; R7 = &y[0]
loop1:  ADD R8, R2, R0      ; R8 = &x[0]
        ADD R9, R3, R0      ; R9 = &y[0]
        ADD R10, R5, R0     ; R10 = n2 (j loop counter)
loop2:  LD  F0, 0(R8)       ; load x[j]
        LD  F2, 0(R6)       ; load x[i]
        FSUB  F0, F0, F2    ; F0 = x[j]-x[i]
        LD  F4, 0(R9)       ; load y[j]
        LD  F6, 0(R7)       ; load y[i]
        FSUB  F4, F4, F6    ; F4 = y[j]-y[i]
        FMUL  F0, F0, F0    ; dx*dx
        FMUL  F4, F4, F4    ; dy*dy
        FADD  F0, F0, F4    ; r2 = dx*dx + dy*dy
        LTD   F0, F30       ; r2 < max (sets FP status bit).
(*)     BFPF  inc2          ; false : skip this next part
        NOP
        CALL  foo           ; Call foo();
        NOP
inc2:   ADDI  R8,R8,#8      ; increment x[j]  pointer
        ADDI  R9,R9,#8      ; increment y[j] pointer
        SUBI  R10, R10, #1
        BNZ   R10, loop2    ; continue with j loop
        NOP
        ADDI  R6,R6,#8      ; increment x[i] pointer
        ADDI  R7,R7,#8      ; increment y[i] pointer
        SUBI  R4,R4,#1      ; decrement i counter
        BNZ   R4, loop1
        NOP
```

For given values of n1 and n2, the inner loop is executed n1*n2 times. The branch at location (*) is taken 65% of the time. Ignore the number of cycles that are executed in the function foo() for now.

Name **Midterm SOLUTIONS**

Part a: (2 points)

How many ALU (integer) instructions, memory instructions, control instructions, and floating point instructions are executed by this code fragment for n1 = n2 = 100? Treat NOP instructions as ALU integer instructions.

ALU (integer) instructions : 2+100*6 + 10000(5) = 50602
Memory instructions: 4*10000 = 40000
Control instructions: 10000*2 + 0.35*10000 + 100 = 23600
Floating Point Operations : 10000*6 = 60000

Part b: (3 points)

Identify all of the hazards in the code and describe how many stall cycles get caused by this hazard. How many stall cycles are due to hazards occur when running this code for n1 = n2 = 100?

1 cycle stall in FSUB F0,F0, F2 due to LD F2, 0(R6)
1 cycle stall in FSUB F4,F4, F6 due to LD F6, 0(R7)
2 cycle stall in FMUL F4,F4, F4 due to FSUB F4,F4,F6 earlier
3 cycle stall in FADD F0,F0,F4 due to FMUL F4,F4,F4
3 cycle stall in LTD F0, F30 due to FADD F0,F0,F4
1 cycle stall in BFPF inc2 due to LTD instruction

Total stall cycles = 11*10000 = 110,000 stall cycles

Part c: (2 points)

What is the total number of cycles spent executing this code? What percentage of cycles are stall cycles? (ignore the instructions executed in foo).

Total cycles = 50602 + 40000 + 23600 + 60000 + 110,000 = 284202 cycles

110,000/284,202 = 39% time spent in stalls.

Part d: (15 points) This has a solution but is the dreaded *Thought* problem.

Optimize the code as much as possible by unrolling the inner loop by 2, rescheduling instructions to eliminate hazards, filling branch delay slots, and eliminating redundant instructions. You are free to rearrange the code anyway you like to get the highest possible performance.

(note : Even Dave Beazley considers this part to be pretty tricky.)

```
          ADD R6, R2, R0      ; R6 = &x[0]
          ADD R7, R3, R0      ; R7 = &y[0]
loop1:    ADD R8, R2, R0      ; R8 = &x[0]
          ADD R9, R3, R0      ; R9 = &y[0]
          ADD R10, R5, R0     ; R10 = n2 (j loop counter)
          LD  F2, 0(R6)       ; load x[i]  (moved out of inner loop)
          LD  F6, 0(R7)       ; load y[i]  (moved out of inner loop)
loop2:    LD  F0, 0(R8)       ; load x[j]
          LD  F4, 0(R9)       ; load y[j]
          FSUB  F0, F0, F2    ; F0 = x[j] - x[i]
          FSUB  F4, F4, F6    ; F4 = y[j] - y[i]
          LD    F10, 8(R8)    ; load x[j+1]
          LD    F14, 8(R9)    ; load y[j+1]
newloop2: FSUB  F10, F10, F2  ; F10 = x[j+1] - x[i]
          FMUL  F20, F0, F0   ; dx*dx
          FMUL  F22, F4, F4   ; dy*dy
          FSUB  F14, F14, F6  ; F14 = y[j+1] - y[i]
          FMUL  F24, F10, F10 ; dx2*dx2
          FADD  F20, F20, F22 ; dx*dx + dy*dy
          ADDI  R8, R8, #16   ; Update x[j] pointer
          ADDI  R9, R9, #16   ; Update y[j] pointer
          FMUL  F26, F14, F14 ; dy2*dy2
          LTD   F20, F30      ; r2 < max??
          SUBI  R10, R10, #2  ; decrement loop counter
          BFPF  test2         ; Skip this next part
          FADD  F24, F24, F26 ; dx2*dx2 + dy2*dy2 (gets executed anyways)
          CALL  foo           ; call foo()
          NOP                 ;  Hard to avoid this
test2:    LD    F0, 0(R8)     ; Load next value x[j]
          LD    F4, 0(R9)     ; Load next value y[j]
          FSUB  F0, F0, F2    ; Do subtraction again
          LTD   F24, F30      ; r2 < max ??
          FSUB  F4, F4, F6    ; Do another subtraction
          BFPF  test3         ; Skip this next part
          LD    F10, 8(R8)    ;
          CALL foo            ; call foo()
          NOP                 ; Not safe to put anything here
test3:    BNZ   R10, newloop2
          LD    F14, 8(R9)
inc2:     ADDI  R6,R6,#8      ; increment x[i] pointer
          SUBI  R4,R4,#1      ; decrement i counter
          BNZ   R4, loop1
          ADDI  R7,R7,#8      ; increment y[i] pointer
```

Part e: (3 points)

How many cycles does it take to execute your new code? How much faster is it than in part (c)?

Cycles = 2 + 100*15 + 5000*(22 + 0.35*4) = 118502 cycles
Speedup = 284202/118502 = 2.40 ==> 140% speedup.