

HOMEWORK III

DUE : October 21, 2009

READ :

- Related portions of Chapters 2, 3 and Appendices A, F and G of the Hennessy book
- Related portions of Chapter 7 of the Jordan book

ASSIGNMENT: There are **four** problems three of which are developed from the Hennessy book.

Solve all homework and exam problems as shown in class and past exam solutions

1) Consider the piece of code studied in Problem 3 of Homework I. This code is for the **DAXPY** application we discussed in class :

```

loop : L.D      F0, 0(R1)      ; load X[i]
      MUL.D    F0, F0, F2     ; multiply a * X[i]
      L.D      F4, 0(R2)     ; load Y[i]
      ADD.D    F0, F0, F4     ; add a * X[i] + Y[i]
      S.D      F0, 0(R2)     ; store Y[i]
      DADDI    R1, R1, #(-8)10 ; decrement X index
      DADDI    R2, R2, #(-8)10 ; decrement Y index
      BNEZ     R1, loop      ; loop if not done

```

Assume that the MIPS is implemented as the **2-way superscalar hardware-speculative** Tomasulo algorithm machine as discussed in class. That is, this is *machine model number 5*. A pair of instructions is issued per cycle, provided that static issuing is preserved. **Two (2)** instructions can be committed in order per cycle provided that they are at the head of the ROB.

Assume also that the functional unit timings are as listed on page A-72 of the Hennessy book : double-precision FP operations ADD.D, MUL.D and DIV.D take 3, 11 and 41 clock periods, respectively ; FP functional units are pipelined and the number of FP reservation station buffers is as given in class ; the number of CDB buses is as given class ; there is a Branch Unit in the EX stage for calculating its effective address and determining the condition ; there is also additional branch prediction hardware in and out of the pipeline ; there are enough functional units for integer instructions not to cause stalls ; the L1 cache memories take **one** clock period each and there are **no** cache misses ; the L1 data cache memory allows two memory accesses for data per clock period if there are no address conflicts.

Assume that there is only **one** iteration. Then, in which clock period, will the **first** iteration of the above loop be completed ? That is, what is the last clock period in which the Commit stage of an instruction from the **first** iteration be done last ? Also, **show** which instructions are flushed out of the pipeline. To answer it, continue the following table :

Instruction	IF	ID	EX	WR	CM
L.D F0, 0(R1)	1	2	3-4	5	6
MUL.D F0, F0, F2	1	2	3/5 - 15	16	17
...Continue...

Indicate any assumptions made during the execution of the loop, if a situation not discussed in class is encountered.

2) Consider the scalar DAXPY code with 11 instructions on page F-9 of the Hennessy book. It is slightly more complex than the scalar DAXPY code shown in Problem 1 above.

Assume that we have the VLIW MIPS processor as discussed in class. That is, this is *machine model number 6*. Additional assumptions are that the latencies are as specified on page 75 of the Hennessy book and that Rx is R1 and Ry is R2.

Show how you would unroll and issue the instructions of this program similar to the VLIW execution sequence given in Figure 2.19 of the Hennessy book. Assume that there are as many vector elements as needed for your unrolling. Note that you will **not** convert the code on F-9 to a high-level code to optimize it to get more parallelism. Just convert the page F-9 code with 11 instructions to a VLIW code.

3) Solve Problem F.2 (b and c) of the Hennessy book. The processor in this question is VMIPS. That is, this is *machine model number 7*. Show the execution timings as discussed in class.

Part (b) has chaining but a **single** memory pipeline connects the CPU to the memory. Part (c) has again chaining, but **three** memory pipelines are provided where three separate memory controllers allow up to three simultaneous pipelined memory accesses if the addresses do not conflict.

4) Solve Problem F.5 (b) of the Hennessy book. The processor in this question is VMIPS. That is, this is *machine model number 7*.

To solve F.5(b), first part (a) has to be solved, resulting in two loops. One of the loops would have a dependence and the other one would have **no** dependence. In part (b), you will convert the loop with **no** dependence to a VMIPS code ! You can use register Ra to point at A[i], and Rb to point at B[i]. Note that the code in the book is in FORTRAN. Its C-like version is as follows :

```

C = 0.0 ;
for (i = 1 ; i <= 64 ; i++)
  { A[i] = A[i] + B[i] ;
    C = C + A[i] ; }

```

RELEVANT QUESTIONS AND ANSWERS

Q1) Consider the following unpipelined MIPS code (*machine model # 0*) where R1 is with the initial value of $(16)_{10}$.

```

loop :  L.D          F2, 0(R1)          ; Statement S1
        ADD.D       F0, F0, F2         ; S2. F0 is initially 0
        DADDI      R1, R1, #(-8)10    ; S3
        BNEZ       R1, loop           ; S4
        S.D        F0, 0(R2)         ; S5
    
```

This code is now run on the 2-way superscalar, statically issued, dynamically scheduled, speculative MIPS. That is, this is *machine model number 5*. A pair of instructions is issued per cycle, provided that static issuing is preserved. **Two** instructions can be committed in order per cycle provided that they are at the head of the ROB.

Assume also that the functional unit timings are as listed on page 75 of the Hennessy book ; FP functional units are pipelined and the number of FP reservation station buffers is as given in class ; the number of CDB buses is as given class ; there is a Branch Unit in the EX stage for calculating its effective address and determining the condition ; there is also additional branch prediction hardware in and out of the pipeline ; there are enough functional units for integer instructions not to cause stalls ; the L1 cache memories take **one** clock period each and there are **no** cache misses ; the L1 data cache memory allows two memory accesses for data per clock period if there are no address conflicts.

Show how the instructions are executed until the loop is completed. When would the execution be complete ? **Show** forwardings and which instructions are flushed out of the pipeline. Indicate any assumptions made during the execution of the loop, if a situation not discussed in class is encountered. To answer it, continue the following table :

			iter #	IF	ID	EX	WR	CM	
loop:	L.D	F2, 0(R1)	1	1	2	3-4	5	6	
	ADD.D	F0, F0, F2	1	1	2	3/5-8	9	10	
		Continue....						

A1) The execution of instructions is shown below. The execution timings are obtained based on the following hardware and software properties of the machine model number 5 :

a) The BNEZ does **not** need any delay slot. We assume that the Branch Unit in EX handles the condition and target address determination while the branch is in the ROB.

b) Additional branch hardware in IF and out of the pipeline is sophisticated enough to make predictions whether a branch has been executed before or not. It inspects instructions for branch possibilities and starts processing them before they are even in the IF stage. It has hardware tables with prediction bits and its own ALU to make address calculations. If a branch is not executed before, it can make a prediction based on the opcode, the offset and any input from the compiler. Thus, when the superscalar MIPS encounters the BNEZ instruction in the above loop for the first time, the prediction is to take the branch since it is a conditional branch with a negative displacement signaling that it could be a loop-ending branch which is almost always taken, except, for example, the last iteration of the loop.

c) The loop requires two iterations, but the branch unit speculates and decides to have the 3rd iteration and more. But, in the 15th clock period, it is realized that the speculation is wrong and so the ROB is flushed and the instruction that follows the BNEZ in the code is fetched (the S.D) in the 16th clock period.

	Instruction	iter #	IF	ID	EX	WR	CM
loop:	L.D F2, 0(R1)	1	1	2	3-4	5	6
	ADD.D F0, F0, F2	1	1	2	3/5-8	9	10
	DADDI R1, R1, #(-8) ₁₀	1	2	3	4	5	6/10
	BNEZ R1, loop	1	2	3	5	6	7/11
loop:	L.D F2, 0(R1)	2	3	4	5-6	7	8/11
	ADD.D F0, F0, F2	2	3	4	5/9-12	13	14
	DADDI R1, R1, #(-8) ₁₀	2	4	5	6	7	8/14
	BNEZ R1, loop	2	4	5	6/7	8	9/15
loop:	L.D F2, 0(R1)	3	5	6	7-8	9	10/15
	ADD.D F0, F0, F2	3	5	6	7/13-15		
	DADDI R1, R1, #(-8) ₁₀	3	6	7	8	9	10/15
	BNEZ R1, loop	3	6	7	8/9	10	11/15
loop:	L.D F2, 0(R1)	4	7	8	9-10	11	12/15
	ADD.D F0, F0, F2	4	7	8	9/15		
	DADDI R1, R1, #(-8) ₁₀	4	8	9	10	11	12/15
	BNEZ R1, loop	4	8	9	10/11	12	13/15
loop:	L.D F2, 0(R1)	5	9	10	11-12	13	14/15
	ADD.D F0, F0, F2	5	9	10	11/15		
	DADDI R1, R1, #(-8) ₁₀	5	10	11	12	13	14/15
	BNEZ R1, loop	5	10	11	12/13	14	15
loop:	L.D F2, 0(R1)	6	11	12	13-14	15	
	ADD.D F0, F0, F2	6	11	12	13/15		
	DADDI R1, R1, #(-8) ₁₀	6	12	13	14	15	
	BNEZ R1, loop	6	12	13	14/15		
loop:	L.D F2, 0(R1)	7	13	14	15		
	ADD.D F0, F0, F2	7	13	14	15		
	DADDI R1, R1, #(-8) ₁₀	7	14	15			
	BNEZ R1, loop	7	14	15			
loop:	L.D F2, 0(R1)	8	15				
	ADD.D F0, F0, F2	8	15				
	S.D F0, 0(R2)		16	17	18	19	20

These instructions are flushed out at the end of the 15th clock period

The execution completes in 20

Q2) Assume that the MIPS is implemented as the **2-way superscalar hardware-speculative** Tomasulo algorithm machine as discussed in class. That is, this is *machine model number 5*. A pair of instructions is issued per cycle, provided that static issuing is preserved. Only **one** instruction can be committed per cycle if it is at the head of the ROB.

Assume also that the functional unit timings are as listed on page 75 of the Hennessy book ; FP functional units are pipelined and the number of FP reservation station buffers is as given in class ; the number of CDB buses is as given class ; there is a Branch Unit in the EX stage for calculating its effective address and determining the condition ; there is also additional branch prediction hardware in and out of the pipeline ; there are enough functional units for integer instructions not to cause stalls ; the L1 cache memories take **one** clock period each and there are **no** cache misses ; the L1 data cache memory allows two memory accesses for data per clock period if there are no address conflicts.

If the **old** MIPS code below (*machine model # 0*) with **two** (2) iterations is run, in which clock period will the second iteration of the loop be completed ? That is, what is the last clock period in which the Commit stage of an instruction from the second iteration is done ? Show which instructions are flushed out of the pipeline. Indicate any assumptions made during the execution of the loop, if a situation not discussed in class is encountered.

```

loop :   L.D      F0, 0(R2)      ; Load from M
        MUL.D   F0, F0, F0     ; M[i] * M[i]
        L.D      F1, 0(R3)     ; Load from N
        L.D      F2, 0(R4)     ; Load from Q
        MUL.D   F3, F1, F2     ; N[i] * Q[i]
        ADD.D   F4, F0, F3     ; M[i] * M[i] + N[i] * Q[i]
        S.D      F4, 0(R1)     ; Store to K
        DADDI   R1, R1, #8     ; Advance the K pointer
        DADDI   R2, R2, #8     ; Advance the M pointer
        DADDI   R3, R3, #8     ; Advance the N pointer
        DADDI   R4, R4, #8     ; Advance the N pointer
        DADDI   R5, R5, #(-1)10 ; Decrement the loop counter
        BNEZ    R5, loop      ; Branch back if not the end
    
```

A2) The execution of the loop for two iterations for the superscalar case is as follows :

Iteration	Instruction	IF	ID	EX	WR	CM	
loop:	1	L.D F0, 0(R2)	1	2	3-4	5	6
	1	MUL.D F0, F0, F0	1	2	3/5-8	9	10
	1	L.D F1, 0(R3)	2	3	4-5	6	7/11
	1	L.D F2, 0(R4)	2	3	4-5	6	7/12
	1	MUL.D F3, F1, F2	3	4	5/6-9	10	11/13
	1	ADD.D F4, F0, F3	3	4	5/10-13	14	15
	1	S.D F4, 0(R1)	4	5	6	7	8/16
	1	DADDI R1, R1, #8	4	5	6	7	8/17
	1	DADDI R2, R2, #8	5	6	7	8	9/18
1	DADDI R3, R3, #8	5	6	7	8	9/19	

	Iteration	Instruction	IF	ID	EX	WR	CM
	1	DADDI R4, R4, #8	6	7	8	9	10/20
	1	DADDI R5, R5, #(-1) ₁₀	6	7	8	9	10/21
	1	BNEZ R5, loop	7	8	9	10	11/22
loop:	2	L.D F0, 0(R2)	7	8	9-10	11	12/23
	2	MUL.D F0, F0, F0	8	9	10/11-14	15	16/24
	2	L.D F1, 0(R3)	8	9	10-11	12	13/25
	2	L.D F2, 0(R4)	9	10	11-12	13	14/26
	2	MUL.D F3, F1, F2	9	10	11/13-16	17	18/27
	2	ADD.D F4, F0, F3	10	11	12/17-20	21	22/28
	2	S.D F4, 0(R1)	10	11	12	13	14/29
	2	DADDI R1, R1, #8	11	12	13	14	15/30
	2	DADDI R2, R2, #8	11	12	13	14	15/31
	2	DADDI R3, R3, #8	12	13	14	15	16/32
	2	DADDI R4, R4, #8	12	13	14	15	16/33
	2	DADDI R5, R5, #(-1) ₁₀	13	14	15	16	17/34
	2	BNEZ R5, loop	13	14	15/16	17	18/35
loop:	3	L.D F0, 0(R2)	14	15	16-17	18	19/35
	3	MUL.D F0, F0, F0	14	15	16/18-21	22	23/35
	3	L.D F1, 0(R3)	15	16	17-18	19	20/35
	3	L.D F2, 0(R4)	15	16	17-18	19	20/35
	3	MUL.D F3, F1, F2	16	17	18/19-22	23	24/35
	3	ADD.D F4, F0, F3	16	17	18/23-26	27	28/35
	3	S.D F4, 0(R1)	17	18	19	20	21/35
	3	DADDI R1, R1, #8	17	18	19	20	21/35
	3	DADDI R2, R2, #8	18	19	20	21	22/35
	3	DADDI R3, R3, #8	18	19	20	21	22/35
	3	DADDI R4, R4, #8	19	20	21	22	23/35
	3	DADDI R5, R5, #(-1) ₁₀	19	20	21	22	23/35
	3	BNEZ R5, loop	20	21	22	23	24/35
loop:	4	L.D F0, 0(R2)	20	21	22-23	24	25/35

These 44 instructions are flushed out at the end of the 35th clock period

Iteration	Instruction	IF	ID	EX	WR	CM
4	MUL.D F0, F0, F0	21	22	23/24-27	28	29/35
4	L.D F1, 0(R3)	21	22	23-24	25	26/35
4	L.D F2, 0(R4)	22	23	24-25	26	27/35
4	MUL.D F3, F1, F2	22	23	24/26-29	30	31/35
4	ADD.D F4, F0, F3	23	24	25/30-33	34	35
4	S.D F4, 0(R1)	23	24	25	26	27/35
4	DADDI R1, R1, #8	24	25	26	27	28/35
4	DADDI R2, R2, #8	24	25	26	27	28/35
4	DADDI R3, R3, #8	25	26	27	28	29/35
4	DADDI R4, R4, #8	25	26	27	28	29/35
4	DADDI R5, R5, #(-1) ₁₀	26	27	28	29	30/35
4	BNEZ R5, loop	26	27	28/29	30	31/35
loop: 5	L.D F0, 0(R2)	27	28	29-30	31	32/35
5	MUL.D F0, F0, F0	27	28	29/31-34	35	
5	L.D F1, 0(R3)	28	29	30-31	32	33/35
5	L.D F2, 0(R4)	28	29	30-31	32	33/35
5	MUL.D F3, F1, F2	29	30	31/32-35		
5	ADD.D F4, F0, F3	29	30	31/35		
5	S.D F4, 0(R1)	30	31	32	33	34/35
5	DADDI R1, R1, #8	30	31	32	33	34/35
5	DADDI R2, R2, #8	31	32	33	34	35
5	DADDI R3, R3, #8	31	32	33	34	35
5	DADDI R4, R4, #8	32	33	34	35	
5	DADDI R5, R5, #(-1) ₁₀	32	33	34	35	
5	BNEZ R5, loop	33	34	35		
loop: 6	L.D F0, 0(R2)	33	34	35		
6	MUL.D F0, F0, F0	34	35			
6	L.D F1, 0(R3)	34	35			
6	L.D F2, 0(R4)	35				
6	MUL.D F3, F1, F2	35				

These 44 instructions are flushed out at the end of the 35th clock period

The two iterations of the loop take 35 clock periods to complete ! 44 instructions are flushed out of the pipeline, as opposed to 11 in the scalar processor case (one instruction issued per clock period).

The superscalar case shows that considerable amount of unfinished work is wasted when the loop completes. This is because multiple issue is used to increase the ILP and it takes considerable time to determine the misspeculation due to committing one instruction per cycle. Thus, committing multiple instructions to match the issue rate is needed. Wasted instructions also means that the memory hierarchy is accessed unnecessarily. This situation would be worse if the FP latencies were longer than 4 clock periods. Thus, an important aspect of superscalar design is to have a high ILP, without increasing the pressure much on the memory hierarchy which can become a bottleneck eventually.

Note that the superscalar execution assumes that there can be two data cache accesses at the same time. For example in the first iteration of the loop, the second and third L.D instructions access the data cache in clock period 5. Since we decided to issue any two instructions at the same, it is plausible to assume that the communication with the data cache has been correspondingly improved to allow two simultaneous accesses.

Overall, we see that speeds of various sections of the pipeline must match to take the full advantage of the new hardware.

Finally, running the old code on the new processor is slow due to RAW data dependencies. A contemporary compiler would have to move the DADDI instructions up and place them between FP instructions.

Q3) Consider the following piece of **old** MIPS code for the unpipelined MIPS processor (*machine model number 0*):

```

loop :   L.D      F0, 0(R2)      ; Load from vector B
          MUL.D   F2, F0, F1    ; F1 is already initialized with "c"
          L.D      F3, 0(R3)    ; Load from vector D
          ADD.D   F4, F2, F3
          S.D      F4, 0(R1)    ; Store to vector A
          DIV.D   F6, F4, F5    ; F5 is already initialized with "e"
          S.D      F6, 0(R3)    ; Store to vector D
          DADDI   R1, R1, #(-8)10 ; Advance the vector A pointer
          DADDI   R2, R2, #(-8)10 ; Advance the vector B pointer
          DADDI   R3, R3, #(-8)10 ; Advance the vector D pointer
          BNEZ    R1, loop      ; Branch back if not the end

```

Assume that the MIPS is implemented as the **2-way superscalar hardware-speculative** Tomasulo algorithm machine as discussed in class. That is, this is *machine model number 5*. A pair of instructions is issued per cycle, provided that static issuing is preserved. **Three (3)** instructions can be committed in order per cycle provided that they are at the head of the ROB.

Assume also that the functional unit timings are as listed on page 75 of the Hennessy book ; FP functional units are pipelined and the number of FP reservation station buffers is as given in class ; the number of CDB buses is as given in class ; there is a Branch Unit in the EX stage for calculating its effective address and determining the condition ; there is also additional branch prediction hardware in and out of the pipeline ; there are enough functional units for integer instructions not to cause stalls ; the L1 cache memories take **one** clock period each and there are **no** cache misses ; the L1 data cache memory allows two memory accesses for data per clock period if there are no address conflicts.

Assume that there are **two** iterations. In which clock period, will the **second** iteration of the loop be completed ? That

is, what is the last clock period in which the Commit stage of an instruction from the **second** iteration be done last ? Also, **show** forwardings but **not** which instructions are flushed out of the pipeline.

A3)

Iteration	Instruction	IF	ID	EX	WR	CM
loop : 1	L.D F0, 0(R2)	1	2	3-4	5	6
1	MUL.D F2, F0, F1	1	2	3/5-8	9	10
1	L.D F3, 0(R3)	2	3	4-5	6	7/10
1	ADD.D F4, F2, F3	2	3	4/9-12	13	14
1	S.D F4, 0(R1)	3	4	5	6	7/14
1	DIV.D F6, F4, F5	3	4	5/13-16	17	18
1	S.D F6, 0(R3)	4	5	6	7	8/18
1	DADDI R1, R1, #(-8) ₁₀	4	5	6	7	8/18
1	DADDI R2, R2, #(-8) ₁₀	5	6	7	8	9/19
1	DADDI R3, R3, #(-8) ₁₀	5	6	7	8	9/19
1	BNEZ R2, loop	6	7	8	9	10/19
loop : 2	L.D F0, 0(R2)	6	7	8-9	10	11/20
2	MUL.D F2, F0, F1	7	8	9/10-13	14	15/20
2	L.D F3, 0(R3)	7	8	9-10	11	12/21
2	ADD.D F4, F2, F3	8	9	10/14-17	18	19/21
2	S.D F4, 0(R1)	8	9	10	11	12/21
2	DIV.D F6, F4, F5	9	10	11/18-21	22	23
2	S.D F6, 0(R3)	9	10	11	12	13/23
2	DADDI R1, R1, #(-8) ₁₀	10	11	12	13	14/23
2	DADDI R2, R2, #(-8) ₁₀	10	11	12	13	14/23
2	DADDI R3, R3, #(-8) ₁₀	11	12	13	14	15/24
2	BNEZ R2, loop	11	12	13	14	15/24

The second iteration of the loop ends at clock period 24.

Q4) Consider the following piece of **old** MIPS code for the unpipelined MIPS processor (*machine model number 0*):

```

loop :  L.D      F2, 0(R2)      ; Load from vector A
        MUL.D   F5, F2, F0     ; F0 is already initialized with “k”
        L.D      F3, 0(R3)     ; Load from vector B
        ADD.D   F6, F3, F1     ; F1 is already initialized with “m”
        L.D      F4, 0(R4)     ; Load from vector C
        DIV.D   F7, F4, F6
        DIV.D   F8, F5, F7
        S.D     F8, 2000(R4)    ; Store to vector D
        DADDI   R2, R2, #8     ; Advance the vector A pointer
        DADDI   R3, R3, #8     ; Advance the vector B pointer
        DADDI   R4, R4, #8     ; Advance the vector C/D pointer
        DADDI   R5, R5, #(-1)10 ; R5 is the loop end counter and initialized to a value
        BNEZ    R5, loop       ; Branch back if not the end

```

Assume that the MIPS is implemented as the **2-way superscalar hardware-speculative** Tomasulo algorithm machine as discussed in class. That is, this is *machine model number 5*. A pair of instructions is issued per cycle, provided that static issuing is preserved. **Two (2)** instructions can be committed in order per cycle provided that they are at the head of the ROB.

Assume also that the functional unit timings are as listed on page 75 of the Hennessy book ; FP functional units are pipelined and the number of FP reservation station buffers is as given in class ; the number of CDB buses is as given in class ; there is a Branch Unit in the EX stage for calculating its effective address and determining the condition ; there is also additional branch prediction hardware in and out of the pipeline ; there are enough functional units for integer instructions not to cause stalls ; the L1 cache memories take **one** clock period each and there are **no** cache misses ; the L1 data cache memory allows two memory accesses for data per clock period if there are no address conflicts.

Assume that there are **two** iterations. In which clock period, will the **second** iteration of the loop be completed ? That is, what is the last clock period in which the Commit stage of an instruction from the second iteration be done last ?

Do **not** show forwardings **nor** the flushed out instructions.

A4)

Iteration	Instruction	IF	ID	EX	WR	CM
loop : 1	L.D F2, 0(R2)	1	2	3-4	5	6
1	MUL.D F5, F2, F0	1	2	3/5-8	9	10
1	L.D F3, 0(R3)	2	3	4-5	6	7/10
1	ADD.D F6, F3, F1	2	3	4/6-9	10	11
1	L.D F4, 0(R4)	3	4	5-6	7	8/11
1	DIV.D F7, F4, F6	3	4	5/10-13	14	15
1	DIV.D F8, F5, F7	4	5	6/14-17	18	19
1	S.D F8, 2000(R4)	4	5	6	7	8/19
1	DADDI R2, R2, #8	5	6	7	8	9/20

loop :

Iteration	Instruction	IF	ID	EX	WR	CM
1	DADDI R3, R3, #8	5	6	7	8	9/20
1	DADDI R4, R4, #8	6	7	8	9	10/21
1	DADDI R5, R5, #(-1) ₁	6	7	8	9	10/21
1	BNEZ R5, loop	7	8	9	10	11/22
2	L.D F2, 0(R2)	7	8	9-10	11	12/22
2	MUL.D F5, F2, F0	8	9	10/11-14	15	16/23
2	L.D F3, 0(R3)	8	9	10-11	12	13/23
2	ADD.D F6, F3, F1	9	10	11/12-15	16	17/24
2	L.D F4, 0(R4)	9	10	11-12	13	14/24
2	DIV.D F7, F4, F6	10	11	12/16-19	20	21/25
2	DIV.D F8, F5, F7	10	1	12/20-23	24	25
2	S.D F8, 2000(R4)	11	12	13	14	15/26
2	DADDI R2, R2, #8	11	12	13	14	15/26
2	DADDI R3, R3, #8	12	13	14	15	16/27
2	DADDI R4, R4, #8	12	13	14	15	16/27
2	DADDI R5, R5, #(-1) ₁	13	14	15	16	17/28
2	BNEZ R5, loop	13	14	15/16	17	18/28

The second iteration of the loop ends at clock period 28.

Q5) A 2-way superscalar MIPS with static issuing and hardware based speculative execution has been designed in class. This is *machine model number 5*. You are asked to modify it to have dynamic issuing. Assume that the latencies are as mentioned on page 75 of the Hennessy book.

How does your design work ? Does dynamic issuing really help the superscalar MIPS ? Elaborate on this by showing cases where sequences of instructions clearly make use of the new issue policy of the MIPS. Be very specific. Is the pressure on the compiler decreased or increased ?

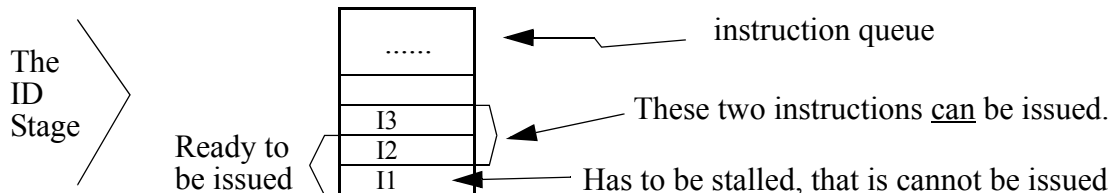
A5) In dynamic issuing if an instruction is stalled in the instruction queue of the ID stage, the instruction(s) behind it can be issued, bypassing the stalled instruction. For the statically issued superscalar MIPS, two instructions are analyzed per clock period in the ID stage : I1 and I2. I1 is stalled if there is no entry in its reservation station and/or no slot in the ROB. I2 is stalled if there is no entry in its reservation station and/or no slot in the ROB. I2 is also stalled if I1 is stalled. We will keep the basic structure used for the superscalar MIPS for easier understanding of the discussion below. The new MIPS still issues two instructions from the instruction queue to the reservation stations and to the reorder buffer per clock period, by keeping in mind that

- i) we should be able to recover from wrong speculations,
- ii) keep precise interrupts and
- iii) detect all potential hazards between an instruction that is issued out-of-order and the instruction(s) it bypasses.

There are four issues that we have to deal with in dynamic issuing :

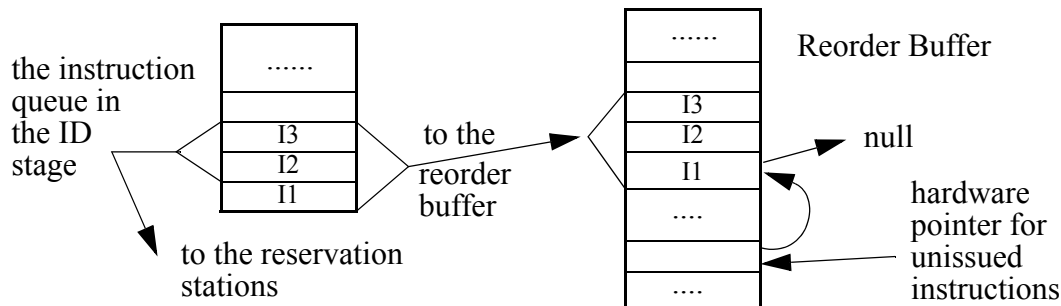
- how do we pick instructions to bypass stalled instructions in the ID stage ?
- where do we keep bypassed instructions ?
- how do we keep track of bypassed instructions ?
- how do we ensure in-order completion ?

Issue (a) : If static issuing is used and the first instruction of the two instructions, I1, is stalled, I2 is also stalled. In dynamic issuing, I2 and I3 can be issued as long as there is no structural hazard associated with them :



Note that if say, I3 cannot be issued with I2, I4 can be issued or I5 or I6. Thus, the ID stage has to have a large “window” of instructions to analyze per clock period, making the ID stage **very** complex. For the rest of the discussion of this problem, assume that I1 cannot be issued, but I2 and I3 can be issued.

Issue (b) : One solution is that we move stalled instructions out of the ID stage (out of the instruction queue) and to the ROB, but treat them as unissued !!! Since I2 and I3 are issued we move them to both the reorder buffer and their reservation stations. We need more connections from ID to reservation stations and the ROB. Also, keeping stalled instructions in the reorder buffer requires that the reorder buffer be **larger** now to allow more instructions :



Issue (c) : To “remember” that there are unissued instructions in the ROB a linked list in hardware can be used. What happens if there are a number of unissued instructions in the ROB “that can be “issued” in addition to the “issuable” instructions in the instruction queue ? We have to give priority to the unissued instructions in the ROB and so we try to move (issue) two instructions from the ROB to the reservation stations. If we cannot, we try to move one instruction from the instruction queue and one from the ROB to the reservation stations. In another scenario, two instructions are issued from the instruction queue. When we move an instruction from the ROB to a reservation station, we “delink” it from the linked list and treat it as issued. The ROB this way satisfies correct speculative execution and precise interrupts. But, the CM stage is much more complex now : we must be able to write from the ROB to the reservation stations after checking that there is no resource dependency, i.e. there is a free reservation station.

Issue (d) : In-order completion is ensured by retiring only those instructions that are at the head of the ROB which are in the order they were stored. Recovery from wrong speculations is done when an incorrectly speculated branch reaches the head of the queue. The instructions behind the branch are flushed out of the ROB. It is possible that some of the flushed instructions are unissued instructions.... Handling precise interrupts is as before : when an instruction reaches the head of the queue its exception is handled. Our solution implies that the CPU will issue two instructions every clock period as long as the ROB is not full. It is guaranteed ! One would try to keep the ROB not filled up, by trying to **commit more than two instructions** per clock period.

Dynamic issuing helps if there are back-to-back long-latency instructions with true dependencies among them. In static issuing, after dependent instructions are issued to their reservation stations, they wait there a long time for long latency instructions. This quickly results in full reservation stations. Structural hazards gradually develop, forcing instructions in the ID stage to stall. Independent instructions behind the stalled instructions in the ID stage stall. Note that this situation can happen if the CPU is running an old code. With dynamic issuing, the pressure on the compiler is reduced since the hardware looks for independent instructions and arbitrarily searches ahead as long as there is space in the reorder buffer. The compiler does not have to do a complete global analysis of the program for best possible sequence of independent instructions. The compiler can be old.

Note also that there is research on out-of-order commit : Mateo Valero and his colleagues published a paper at the High Performance Computer Architecture (HPCA) conference in 2004 : “Out-of-Order Commit Processors.”

Q6) Write the MIPS code for a *machine model number 2* assuming latencies given on page 75 of the Hennessy book for the following piece of high-level code :

```

k = 0.0
for (i = 1 ; i <= 64 ; i++)
    k = k + A[i] * B[i]

```

Then, assuming that the MIPS is a VLIW machine as described in class. That is, this is *machine model number 6*. Show how the instructions are issued for each clock period. Your code will be again based on the latencies as specified on page 75 of the Hennessy book.

A6) This code implements the dot product. The scalar code for this piece of high-level language code is as follows :

```

loop :      L.D          F0, #0
           L.D          F2, 0(R8)          ; R8 points at the end of vector A
           L.D          F4, 0(R9)          ; R9 points at the end of vector B
           DADDI        R9, R9, #(-8)10
           MUL.D        F6, F2, F4
           NOP
           DADDI        R8, R8, #(-8)10
           BNEZ         R8, loop
           ADD.D        F0, F0, F6
           NOP
           NOP
           S.D          F0, 0(R10)         ; R10 points at scalar “k”

```

We unroll the loop eight times in order to have as many operations as possible during the VLIW execution. After the loop is exited we have to do more computations : four more instructions (three adds and a store) as shown on the next page. Though, it seems there are a lot of NOPs after we exit the loop, it is **not** the case : instructions out of the loop are placed around these four instructions.

Note also the importance of having many registers in the architecture for effective VLIW execution. In this case, having many FP registers helps unroll the loop eight times :

	Mem 1	Mem 2	FP 1	FP 2	Int/Br
	L.D F0, #0	L.D F2, #0			
loop	L.D F4, #0	L.D F6, #0			
↙	L.D F8, 0(R8)	L.D F10, 0(R9)			
	L.D F12, (-8) ₁₀ (R8)	L.D F14, (-8) ₁₀ (R9)			
	L.D F16, (-16) ₁₀ (R8)	L.D F18, (-16) ₁₀ (R9)	MUL.D F8, F8, F10		
	L.D F20, (-24) ₁₀ (R8)	L.D F22, (-24) ₁₀ (R9)	MUL.D F12, F12, F14		
	L.D F24, (-32) ₁₀ (R8)	L.D F26, (-32) ₁₀ (R9)	MUL.D F16, F16, F18		
	L.D F28, (-40) ₁₀ (R8)	L.D F30, (-40) ₁₀ (R9)	MUL.D F20, F20, F22		
	L.D F8, (-48) ₁₀ (R8)	L.D F10, (-48) ₁₀ (R9)	MUL.D F24, F24, F26	ADD.D F0, F0, F8	
	L.D F12, (-54) ₁₀ (R8)	L.D F14, (-54) ₁₀ (R9)	MUL.D F28, F28, F30	ADD.D F2, F2, F12	
			MUL.D F8, F8, F10	ADD.D F4, F4, F16	
			MUL.D F12, F12, F14	ADD.D F6, F6, F20	
				ADD.D F0, F0, F24	
				ADD.D F2, F2, F28	DADDI R8, R8, #(-64) ₁₀
				ADD.D F4, F4, F8	DADDI R9, R9, #(-64) ₁₀
end of the loop				ADD.D F6, F6, F12	BNEZ R8, Loop
↙	←				Branch delay slot
			ADD.D F0, F0, F2		
			ADD.D F4, F4, F6		
			ADD.D F0, F0, F4		
	S.D F0, 0(R10)				

Q7) Consider the old **original** MIPS code in Past Exam Question 4 again.

Assume that the MIPS is implemented as the **VLIW MIPS** processor : That is, this is *machine model number 6*. Assume that the functional unit timings are as listed in **Figure 2.2** on page 75 of the Hennessy book.

Show how you would unroll and issue the instructions as done in class. Assume that there are as many iterations as needed for your unrolling.

A7) The loop is unrolled four (4) times. The VLIW MIPS code is as follows :

	Mem 1	Mem 2	FP 1	FP 2	Int/Br
loop :	L.D F6, 0(R3)	L.D F7, 8(R3)			
	L.D F2, 0(R2)	L.D F8, (16) ₁₀ (R3)			
	L.D F3, 8(R2)	L.D F9, (24) ₁₀ (R3)	ADD.D F18, F6, F1	ADD.D F19, F7, F1	
	L.D F4, (16) ₁₀ (R2)	L.D F10, 0(R4)	MUL.D F14, F2, F0	ADD.D F20, F8, F1	
	L.D F5, (24) ₁₀ (R2)	L.D F11, (8) ₁₀ (R4)	MUL.D F15, F3, F0	ADD.D F21, F9, F1	
		L.D F12, (16) ₁₀ (R4)	MUL.D F16, F4, F0		
		L.D F13, (24) ₁₀ (R4)	MUL.D F17, F5, F0	DIV.D F22, F10, F18	
			DIV.D F23, F11, F19	DIV.D F24, F12, F20	
				DIV.D F25, F13, F21	
			DIV.D F26, F14, F22		
			DIV.D F27, F15, F23	DIV.D F28, F16, F24	DADDI R2, R2, #(32) ₁₀
			DIV.D F29, F17, F25		DADDI R3, R3, #(32) ₁₀
		S.D F26, 2000(R4)			DADDI R5, R5, #(-4) ₁₀
		S.D F27, 2008(R4)	S.D F28, 2010(R4)		BNEZ R5, Loop
end of the loop →	S.D F29, 2018(R4)			DADDI R4, R4, #(32) ₁₀ ← Branch delay slot	

Q8) Consider the past exam question Q5 of Homework 2. Assume that the MIPS is implemented as the **VLIW MIPS** processor : That is, this is *machine model number 6*. Assume that the functional unit timings are as listed in **Figure 2.2** on page 75 of the Hennessy book.

Show how you would unroll and issue the instructions as done in class. Assume that there are as many iterations as needed for your unrolling.

Hint : Unroll the loop **eight (8)** times !

A8) The VLIW MIPS code is below. The loop is unrolled **eight (8)** times :

	Mem 1	Mem 2	FP 1	FP 2	Int/Br
loop :	L.D F1, 0(R1)	L.D F0, 8(R1)			
	L.D F6, (16) ₁₀ (R1)	L.D F7, (24) ₁₀ (R1)			
	L.D F8, (32) ₁₀ (R1)	L.D F9, (40) ₁₀ (R1)	ADD.D F12, F2, F1	ADD.D F13, F2, F0	
	L.D F10, (48) ₁₀ (R1)	L.D F11, (56) ₁₀ (R1)	ADD.D F14, F2, F6	ADD.D F15, F2, F7	
	L.D F4, 0(R4)	L.D F20, 8(R4)	ADD.D F16, F2, F8	ADD.D F17, F2, F9	
	L.D F21, (16) ₁₀ (R4)	L.D F22, (24) ₁₀ (R4)	ADD.D F18, F2, F10	ADD.D F19, F2, F11	
	L.D F23, (32) ₁₀ (R4)	L.D F24, (40) ₁₀ (R4)	SUB.D F12, F4, F12	SUB.D F13, F20, F13	
	L.D F25, (48) ₁₀ (R4)	L.D F26, (56) ₁₀ (R4)	SUB.D F14, F21, F14	SUB.D F15, F22, F15	
	S.D F12, 0(R1)	S.D F13, 8(R1)	SUB.D F16, F23, F16	SUB.D F17, F24, F17	
	S.D F14, (16) ₁₀ (R1)	S.D F15, (24) ₁₀ (R1)	SUB.D F18, F25, F18	SUB.D F19, F26, F19	
	S.D F16, (32) ₁₀ (R1)	S.D F17, (40) ₁₀ (R1)	MUL.D F1, F12, F1	MUL.D F0, F13, F0	
S.D F18, (48) ₁₀ (R1)	S.D F19, (56) ₁₀ (R1)	MUL.D F6, F14, F6	MUL.D F7, F15, F7		
		MUL.D F8, F16, F8	MUL.D F9, F17, F9	DADDI R1, R1, #(64) ₁₀	
S.D F1, 0(R5)	S.D F0, 8(R5)	MUL.D F10, F18, F9	MUL.D F11, F19, F11	DADDI R4, R4, #(64) ₁₀	
S.D F6, (16) ₁₀ (R5)	S.D F7, (24) ₁₀ (R5)			DADDI R6, R6, #(-8) ₁₀	
S.D F8, (32) ₁₀ (R5)	S.D F9, (40) ₁₀ (R5)			BNEZ R6, loop	
S.D F10, (48) ₁₀ (R5)	S.D F11, (56) ₁₀ (R5)			DADDI R5, R5, #(64) ₁₀	

end of the loop

Branch delay slot

Q9) Consider the following VMIPS code :

```

L.D      F0, R1      ; R1 points at variable "a"
LV       V1, R2      ; R2 points at array X
MULVS.D V2, V1, F0
LV       V3, R3      ; R3 points at array Y
ADDV.D  V4, V2, V3
SV       R3, V4

```

a) Write down the corresponding algorithm in the style used in class. Assume that the VLR and VM registers are initialized to allow 64-element vector operations. What is the time complexity of the algorithm ?

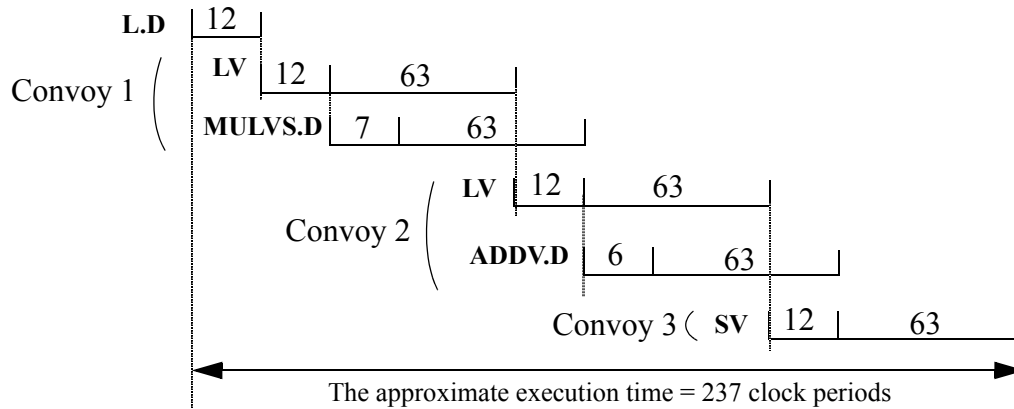
b) Determine how many clock periods it would take to run the above VMIPS vector code. This is *machine model number 7*. Assume that the hardware allows chaining but there is only one memory pipeline. Indicate any assumptions made during the execution of the loop, if a situation not discussed in class is encountered.

A9) a) for (i = 1 ; i <= 64 ; i++)

$$Y[i] = (a * X[i]) + Y[i] \Big) O(n)$$

There is a single loop with one statement. The loop iterates n times, the length of the vectors. The time complexity is then linear : $O(n)$.

b) We observe that due to one memory pipeline there are three convoys. Then the execution time is :



Q10) Consider the following code :

```
k = 0.0 ;
for (i = 1 ; i <= 64 ; i++)
    k = k + A[i] * B[i]
```

a) Write the above code in terms of VMIPS instructions. That is, this is *machine model number 7*.

Indicate the approximate execution time for your code, assuming that there is **no** chaining, the MIPS is 2-way super-scalar with scalar FP, committing 1 instruction per clock period. Scalar integer pipeline latencies are as given on page 75 of the Hennessy book and the branch latency is 1, except that the L.D and S.D take 12 clock periods each.

Indicate any assumptions made during the execution of the loop, if a situation not discussed in class is encountered.

b) Then, invent and describe new vector instructions for the VMIPS to speed up the processor and mention the new approximate execution time. Is there any major hardware change in the VMIPS now ?

Indicate any assumptions made during the execution of the loop, if a situation not discussed in class is encountered.

A10) a) This piece of code implements the “dot product”.

We will use the loop fission technique mentioned in question F.5 of the Hennessy book to obtain the machine code to have a loop with dependence and another with no dependence to have a higher speed :

```
k = 0.0 ;
for (i = 1 ; i <= 64 ; i++)
    dot[i] = A[i] * B[i] ;
for (i = 1 ; i <= 64 ; i++)
    k = k + dot[i] ;
```

The VMIPS code is as follows :

```

                LV          V1, Ra          ; time : 12 + 63
                LV          V2, Rb          ; 12 + 63
                MULV.D      V3, V2, V1      ; 7 + 63
                SV          Rdot, V3        ; 12 + 63
loop :          L.D         F0, #0         ; 12
                L.D         F2, 0(Rd)      ; instructions
                DADDI       Rd, Rd, #(-8)10 ; in the loop
                ADD.D       F0, F0, F2     ; take
                BNEZ        Rd, loop       ; 4 * 64
                S.D         F0, Rk         ; 12 + 1 (ADD.D to S.D latency delay)

```

“Rd” is for the dot vector just like the Rdot register is. But, Rd initially points at the last element of the dot vector, not at the first element of the dot vector. The “L.D F0, #0” instruction takes 12 clock periods then the instructions in the loop below it are completed every clock period after. Note that in this second loop, there is no vector instruction !!! The timing assumes there is no chaining. The approximate number of clock periods is 582.

Note that the same application (dot-product) is studied in the previous Past Exam question (Q6 and A6 above) for the VLIW MIPS. There, the approximate execution time is 132 clock periods. Why such a large difference ? First, the latencies of VLIW instructions are shorter than those of VMIPS instructions. Second, for the VMIPS, the lack of chaining increases the execution time by limiting the overlapping of vector instructions. Third, the lack of multiple memory pipelines limits simultaneous independent vector loads and stores to proceed. Finally, the lack of overlapping of vector instruction executions with the scalar instruction executions in the loop forces the loop to start after all vector instructions are completed. If these limitations are removed, which would be the case for a supercomputer, the approximate execution time for the VMIPS would be approximately 245 clock periods.

Still the time is not better than the VLIW case !! However, the VLIW case suffers from two problems not observed in the VMIPS case and can make the VLIW slower : the VLIW code is large with 3240 bits (which also includes instructions not related to the dot product operation). The VMIPS code is 384 bits. The second and more important problem is that the memory interface unit of VLIW machines is not as sophisticated as that of vector machines. Thus, there will be a lot of idle clock periods for fetching so many VLIW instruction bits and also performing two parallel data accesses per clock period. The VLIW execution time would be worse than 132 clock periods...

b) The dot-product is a “reduction” operation where two vectors (A and B) are reduced to a scalar number “k.” So, we devise a new VMIPS vector instruction called DPV (Dot Product of Vectors) :

```
DPV    F0, V1, V2          ; F0 <--- V1 * V2
```

Assume that the latency for the DPV is 10 clock periods, since after the multiplication an addition is needed. Then the dot product program becomes :

```

LV      V1, Ra          ; time : 12 + 63
LV      V2, Rb          ; 12 + 63
DPV     F0, V1, V2      ; 10 + 63
S.D     F0, Rk          ; 12

```

The approximate time
is 235 clock periods.

The speedup is 2.18

The hardware is changed such that the vector FP multiplier unit is chained to the scalar FP add unit (which stores in a scalar FP register). The chaining requires a new bus, a Vector-Scalar, VS bus, from the vector multiplier to the scalar FP Add functional unit. In the context of speculative superscalar MIPS execution, this means a reservation station entry for the DPV in the integer section. That reservation station waits for the 64 results coming on the VS bus from the vector unit. The VS bus is parallel to the integer and FP CDB bus(es).

If full chaining and multiple memory pipes are employed, the execution time would be 109 clock periods.

Q11) Consider the old **original** MIPS code in Past Exam Question 4 again. Assume that the MIPS is implemented as the **VMIPS** processor : That is, this is *machine model number 7*.

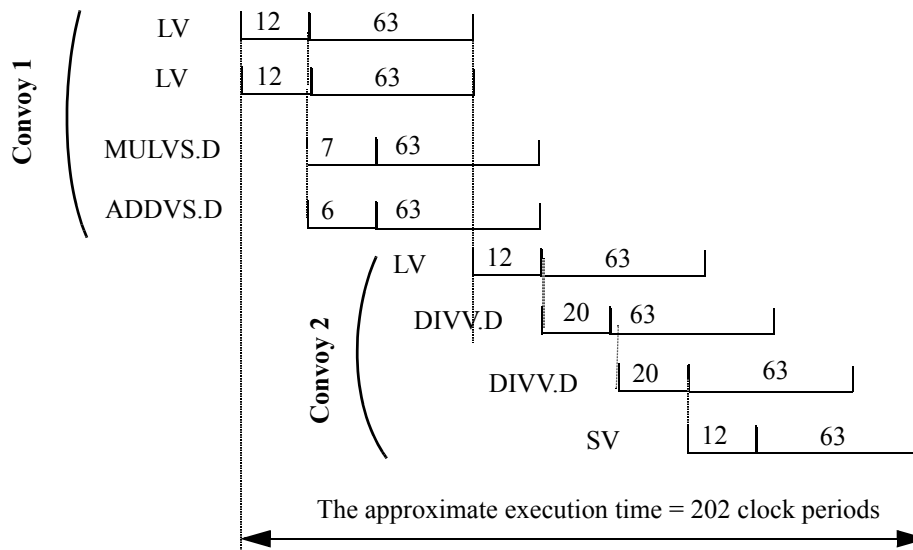
a) Rewrite the old code in terms of VMIPS instructions. Add comments to your code. Assume that the loop has **64** iterations. Also assume that the VM, VLR, R2, R3, R4, F0 and F1 registers have been already appropriately initialized. Finally, assume that R6 is initialized to (R4 + 2000).

b) Show the execution timings as discussed in class. Assume that there is **chaining**. Also assume that there are **two (2)** memory pipelines.

A11) a) The VMIPS code is as follows :

LV	V1, R2	; Load vector A to V1
LV	V2, R3	; Load vector B to V2
MULVS.D	V3, V1, F0	; Multiply each element of vector A by “k”
ADDVS.D	V4, V2, F1	; Add each element of vector B by “m”
LV	V5, R4	; Load vector C to V5
DIVV.D	V6, V5, V4	; Divide each element of vector C by the result of the add
DIVV.D	V7, V3, V6	; Divide the result of the multiply by the result of the divide
SV	R6, V7	; Store the result to vector D

b) The execution timing is as follows :



Q12) Consider the old MIPS code obtained in Past Exam Question 2 above again. The processor in this question is VMIPS. That is, this is *machine model number 7*. The VMIPS vector and scalar pipeline timings are as given on page F-13 and 75 of the Hennessy book, respectively. Assume that there is **no** chaining. Also assume that there is **one** memory pipeline. Finally, assume that the VM and VLR registers are already initialized.

Rewrite the old MIPS code in terms of VMIPS instructions. Show the execution timings as discussed in class. Assume that the loop has **64** iterations. Indicate any assumptions made during the execution of the loop, if a situation not discussed in class is encountered.

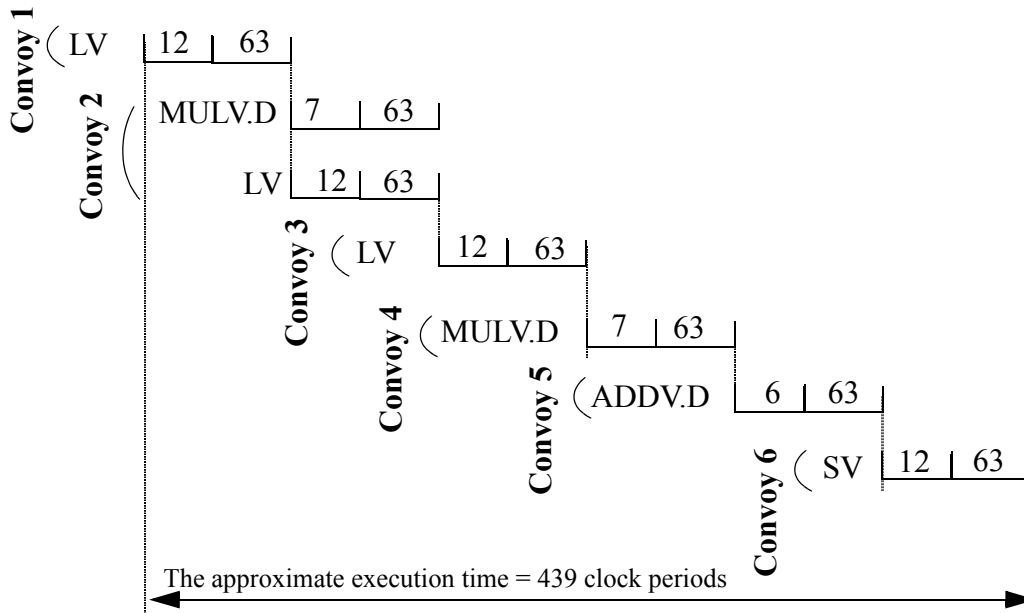
A12) The VMIPS code is as follows :

```

LV      V1, R2      ; Load from M
MULV.D V1, V1, V1   ; M[i] * M[i]
LV      V2, R3      ; Load N
LV      V3, R4      ; Load Q
MULV.D V4, V2, V3   ; N[i] * Q[i]
ADDV.D V5, V4, V1   ; M[i] * M[i] + N[i] * Q[i]
SV      R1, V5      ; Store to K
    
```

Since there are only 64 iterations and the associated registers are already initialized, the vectorized code does not have any scalar instruction above.

The timing of the vectorized code execution is as follows :



Q13) Consider the following piece of **old** MIPS code written for its unpipelined version :

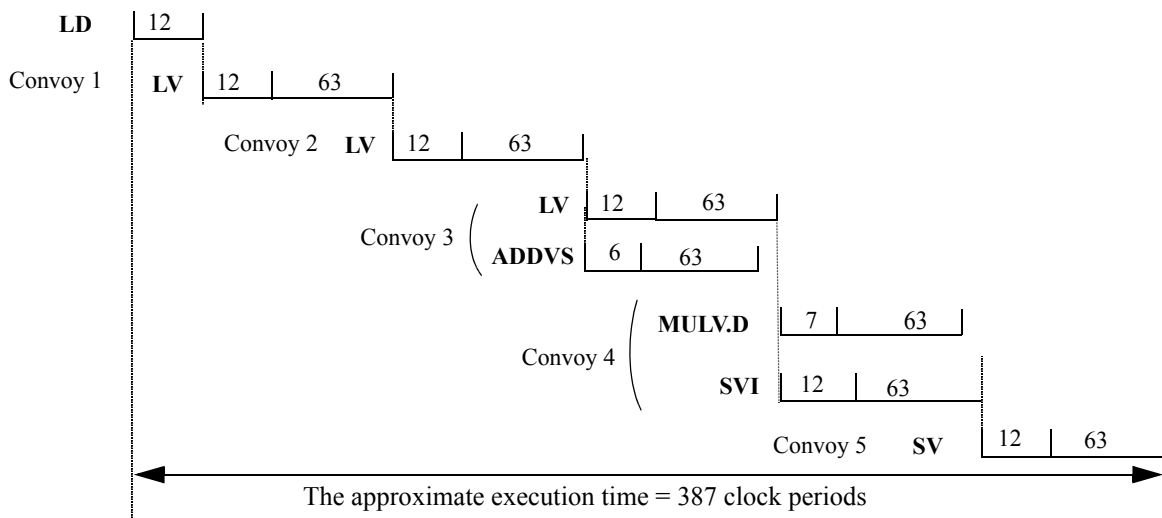
```

          DADDI    R1, R0, #(64)10      ; Memory accesses are commented below :
          L.D      F0, 0(Rk)              ; Rk points at constant k
loop:    LD        Ra, 0(Rindexa)         ; Rindexa points at index vector for vector A
          L.D      F2, 0(Rb)             ; Rb points at vector B
          L.D      F4, 0(Rd)             ; Rd points at vector D
          ADD.D    F6, F2, F0
          MUL.D    F8, F6, F4
          S.D      F6, 0(Ra)              ; Stores to vector A
          S.D      F8, 0(Rc)              ; Stores to vector C
          DADDI    R1, R1, #(-1)10
          DADDI    Rindexa, Rindexa, #4   ; Rindexa is advanced
          DADDI    Rb, Rb, #8             ; Rb is advanced
          DADDI    Rc, Rc, #8             ; Rc is advanced
          DADDI    Rd, Rd, #8             ; Rd is advanced
          BNEZ    R1, loop
    
```

Write that code in terms of VMIPS instructions (**vectorize it**). This is *machine model number 7*. Indicate the approximate execution time for your code, assuming that there is **no** chaining, a single memory pipeline and the MIPS scalar FP and scalar integer pipeline latencies are as given on page 75 of the Hennessy book. Indicate any assumptions made during the execution of the loop, if a situation not discussed in class is encountered.

A13)

LD	F0, 0(Rk)	; 12
LV	V _p , Rindexa	; Convoy 1
LV	V _b , Rb	; Convoy 2
LV	V _d , Rd	; Convoy 3
ADDVS.D	V _a , V _b , F0	; Convoy 3
MULV.D	V _c , V _a , V _d	; Convoy 4
SVI	(R0 + V _p), V _a	; Convoy 4
SV	Rc, V _c	; Convoy 5



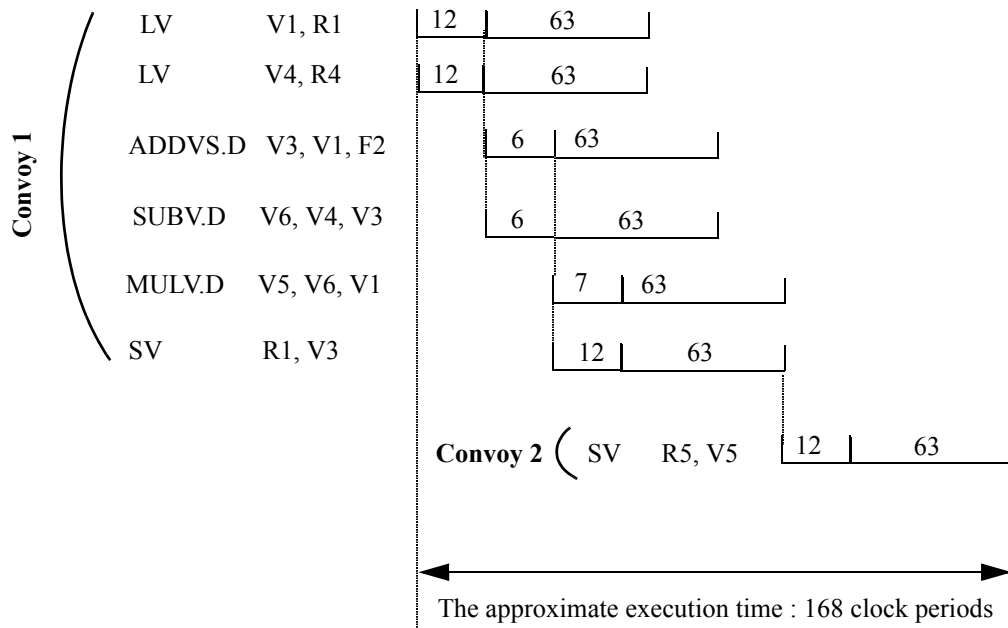
Q14) Consider the past exam question Q5 of Homework 2. Assume that the MIPS is implemented as the **VMIPS** processor : That is, this is *machine model number 7*.

- a) **Rewrite** the old code in terms of VMIPS instructions. Add comments to your code. Assume that the loop has **64** iterations. Also assume that the VM, VLR, R1, R4, R5 and F2 registers have been already appropriately initialized.
- b) **Show** the execution timings as discussed in class. Assume that there is **chaining**. Also assume that there are **three (3)** memory pipelines.

A14) a) The VMIPS code is as follows :

LV	V1, R1	; Load vector A to V1
LV	V4, R4	; Load vector B to V2
ADDVS.D	V3, V1, F2	; Add "k" to each element of vector A
SUBV.D	V6, V4, V3	; Subtract the new value of each element of A from each element of B
MULV.D	V5, V6, V1	; Multiply each subtraction result by an old element of A
SV	R1, V3	; Store the result back to vector A
SV	R5, V5	; Store the multiplication result to vector C

b) The execution timing is as follows :



Q15) Superscalar processors have closed the speed gap they had with vector processors considerably when they run vector-oriented applications with loop-level parallelism. One main advantage of vector processors that still remains, is their handling of pipelined non-unit stride and sparse matrix memory accesses.

Suggest hardware and software techniques that can help **superscalar** processor close this gap too. You may invent your own reasonable solutions.

A15) A number of software techniques can be tried on the scalar code so that

- i) the access pattern of the code is changed from rows/columns to blocks (blocking). That is, the code accesses a block containing a portion of a number of rows and columns. In order to do that arrays are stored blockwise, not row-major nor column-major (see Blocking on page 303 of the Hennessy book).
- ii) back-to-back nonsequential memory accesses are converted to sequential memory accesses : loop interchange (see Loop Interchange on page 302 of the Hennessy book).
- iii) data brought into the cache is used again and again without misses then removed from the cache : loop fusion (**not** loop fission).
- iv) The compiler unrolls loops to access elements in advance. Also, for non-loop situations, the compiler rearranges the code so that memory reference instructions are moved up and down.

A number of hardware+software techniques can be used :

- i) Special buffers between the memory and the cache can prefetch data automatically. If there are special data prefetch instructions for caches, the compiler inserts them appropriately.
- ii) The data cache can deliver in a pipelined fashion where the first memory request to a cache block is checked for TLB and cache hits, taking more than one clock period. After that, in every clock period an element from the same block is delivered : TLB translation and cache hit checking are done only once, for example, 64 data elements. Note that this is an active research area now.
- iii) Second and third level cache memories can perform similarly.
- iv) There can be indexed memory access instructions for cache memories