计算机系统结构复习题（1）： 指令级并行

一、简答题

1. 简述 VLIW（超长指令字）处理器和 SuperScalar(超标量)处理器的区别。

2. 什么是精确中断？如果有一条指令在执行阶段发生了两种类型的异常： 指令地址（取指）异常和 ALU 溢出， 那么哪一个异常会作为指令中断原因记录？ 提示：以五阶段流水线处理器作为流水线模型进行回答。

3. 指出下列指令序列中的数据相关性。

1) Mult $4, $3, $6
2) Add $5, $3, $7
3) Store $5, 0($8)
4) Load $2, 100($8)
5) Mult $1,$2,$3
6) Add $3, $1, $2
7) Add $3, $5, $6

指令2与指令3对于$5存在写后读 （ RAW)相关性
指令4与指令5对于$2存在写后读 （ RAW)相关性
指令5与指令6对于$1存在写后读 （ RAW)相关性
指令5与指令6对于$3存在读后写 （ WAR)相关性
指令6与指令7对于$3存在写后写 （ WAW)相关性

二、综合分析题

1. **调度问题** Scheduling

In this problem, we examine the execution of a code segment on in-order and out-of order processors. The code we consider scales a vector of floating-point numbers by a constant.

```
I1      loop: LD.D F0, 0(R1)
I2            MUL.D F0, F2, F0
I3            ST.D F0, 0(R1)
I4            DDI R1, R1, 8
I5            BNE R1, R2, loop
```

Problem 1.A

First, we consider the fully-bypassed, single-issue, in-order pipeline in Figure1. Load results are available at the end of the X3 stage, and floating-point multiply results are available at the end of the X4 stage. Stores don't need their data until the X2 stage. Assume perfect branch target prediction (i.e. there are no bubbles

due to branches.)

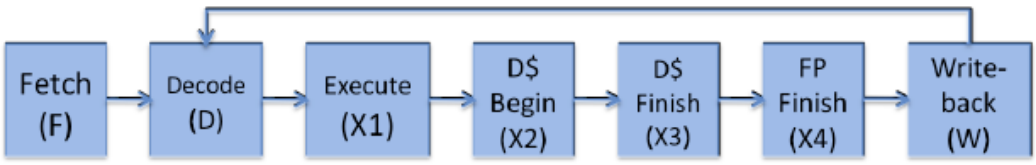


Figure 1. In-order pipeline. Full bypassing is not shown.

Fill in Table 1 to indicate how the given code will execute for two loop iterations. In each cell, write the name of the stage that the instruction is currently in. The first two rows have been completed for you.

Inst \ Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
$I_{1,\text{ iteration 1}}$	F	D	X1	X2	X3	X4	W																							
$I_{2,\text{ iteration 1}}$		F	D	D	D	X1	X2	X3	X4	W																				
$I_{3,\text{ iteration 1}}$			F	F	F	D	D	D	X1	X2	X3	X4	W																	
$I_{4,\text{ iteration 1}}$						F	F	F	D	X1	X2	X3	X4	W																
$I_{5,\text{ iteration 1}}$									F	D	X1	X2	X3	X4	W															
$I_{1,\text{ iteration 2}}$										F	D	X1	X2	X3	X4	W														
$I_{2,\text{ iteration 2}}$											F	D	D	D	X1	X2	X3	X4	W											
$I_{3,\text{ iteration 2}}$												F	F	F	D	D	D	X1	X2	X3	X4	W								
$I_{4,\text{ iteration 2}}$															F	F	F	D	X1	X2	X3	X4	W							
$I_{5,\text{ iteration 2}}$																		F	D	X1	X2	X3	X4	W						

Table 1. Code schedule for the in-order processor in Q. 1.A.

Problem 1.B

Averaged over a large number of iterations, what is the throughput of the loop executing on the processor in Problem 1.A, measured in cycles per iteration?

9 cycles per iteration.
The write back of I5 for the second iteration is 9 cycles after the write back of I5 for the first iteration. This pattern will continue, so we'll average 9 cycles per iteration.

Problem 1.C

Next, we consider an idealized single-issue, out-of-order pipeline without register renaming, shown in Figure 2. Instructions can issue out-of-order once no RAW, WAW, or WAR hazards exist. The issue stage can hold an unbounded number of waiting instructions. As with Q1.A, the pipeline is fully bypassed, load results are available at the end of X3, multiply results are available at the end of X4, store data is consumed in the X2 stage, and branch target prediction is perfect. Write backs can occur out of order.

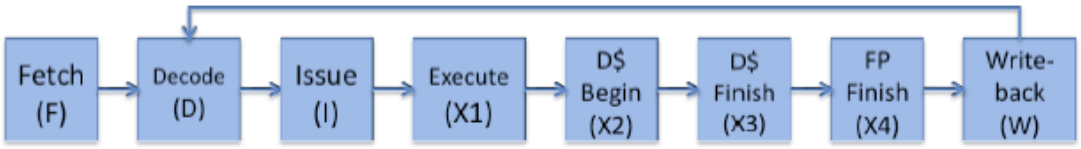


Figure 2. Out-of-order pipeline. Full bypassing is not shown.

Fill in Table 2 to indicate how the given code will execute for two loop iterations. In each cell, write the name of the stage that the instruction is currently in. The first two rows have been completed for you.

Inst \ Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
$I_{1,\text{ iteration 1}}$	F	D	I	X1	X2	X3	X4	W																						
$I_{2,\text{ iteration 1}}$		F	D	I	I	I	X1	X2	X3	X4	W																			
$I_{3,\text{ iteration 1}}$			F	D	I	I	I	I	I	X1	X2	X3	X4	W																
$I_{4,\text{ iteration 1}}$				F	D	I	I	X1	X2	X3	X4	W																		
$I_{5,\text{ iteration 1}}$					F	D	I	I	X1	X2	X3	X4	W																	
$I_{1,\text{ iteration 2}}$						F	D	I	I	I	X1	X2	X3	X4	W															
$I_{2,\text{ iteration 2}}$							F	D	I	I	I	I	I	X1	X2	X3	X4	W												
$I_{3,\text{ iteration 2}}$								F	D	I	I	I	I	I	I	I	X1	X2	X3	X4	W									
$I_{4,\text{ iteration 2}}$									F	D	I	X1	X2	X3	X4	W														
$I_{5,\text{ iteration 2}}$										F	D	I	X1	X2	X3	X4	W													

Table 2. Code schedule for the out-of-order processor in Q 1.C.

Problem 1.D

Averaged over a large number of iterations, what is the throughput of the loop executing on the processor in Q 1.C, measured in cycles per iteration?

Without renaming, the bottleneck will be the latency from the issue of the load to the issue of the store (inclusive), which is 7 cycles.

Problem 1.E

Finally, consider a pipeline that is identical to the one in Q1.C, except that it now has register renaming with an unbounded number of physical registers. Averaged over a large number of iterations, what is the throughput of the loop executing on this processor, measured in cycles per iteration? (Hint: it may not be necessary to manually schedule the code to determine the answer to this question.)

Now, only RAW hazards matter, so we will get multiple iterations of the loop in-flight at a time, allowing us to sustain a CPI of 1.

____5_____ Cycles per Iteration

2: Branch Prediction in the Pipeline

Ben is designing a 7-stage in-order pipeline and is concerned about the performance implications of branches. The baseline processor he is considering is similar to the classic 5-stage RISC pipeline, except instruction cache access and data cache access each take two stages, as shown in Figure 2-1. Initially, the pipeline lacks any branch prediction mechanism. All branches in Ben's ISA are simple enough that they can execute without an ALU. His ISA has no branch delay slots.

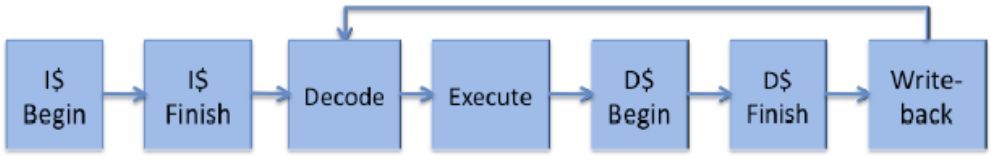


Figure 2-1 Ben's in-order pipeline

Problem 2.A

What is the earliest stage that branches can be resolved in Ben's pipeline? How many instructions are squashed on a taken branch? Assuming 1 in 6 instructions is a branch and 3/5 branches are taken, and assuming a base CPI of 1, what is the CPI of Ben's pipeline?

Simple branches can be resolved in decode, so two fetches are wasted on a taken branch. The CPI is 5/6 (non-branches) + 1/6*2/5 (untaken branches) + 3*1/6*3/5 (taken branches), or 6/5.

___Decode____ Branch Resolution Stage
_____2_____ # Instructions Squashed
_____6/5_____ CPI

Problem 2.B

Ben is considering adding a branch history table (BHT) to predict branch outcomes. In what pipeline stage would the BHT's prediction be acted upon? In what situations will the BHT improve performance?

The BHT is untagged, so its prediction shouldn't be acted upon until the instruction is known to be a branch (i.e. in the decode stage). Thus, the BHT usually doesn't help at all, unless a RAW hazard prevents it from resolving in the decode stage—namely, if a load precedes a dependent branch.

_____Decode___ BHT Prediction Stage

Problem 2.C

Ben considers adding a single-cycle latency branch target buffer (BTB) as an alternative to the BHT. In what pipeline stage would the BTB produce a next-PC prediction? If the BTB predicts taken branches correctly 80% of the time and mistakenly predicts not-taken branches as taken 20% of the time, what is Ben's new CPI, assuming a base CPI of 1?

The BTB is indexed by the current PC, so its prediction can be used as soon as the PC is known, during the I$ Begin stage. There is no bubble on taken branches now (assuming a correct prediction); mis-predicted branches (either taken or not-taken) will incur a 2-cycle penalty. So the CPI is 5/6 +1/6*0.8 (correctly-predicted branches) + 3*1/6*0.2 (mispredicted branches), or 16/15.

_____I$ Begin_____ BTB Prediction Stage， _____16/15_____ CPI

Problem 2.D

Ben's original BTB design consisted of a next-PC prediction, a tag, and a valid bit per entry. The BTB's prediction was only used if the valid bit was set and the tag matched the current PC. Ben wishes to keep the BTB's behavior exactly the same, but save silicon area by omitting the valid bit. How can he accomplish this goal?

The important insight is that when we predict that a branch is not-taken, we're predicting that the next PC is PC+4. So, for predicted-not-taken branches (or non-branches), we can set the next-PC prediction to PC+4 instead of clearing the valid bit.

Problem 3: Potpourri

Problem 3.A

Describe how precise exceptions are maintained in out-of-order processors.

Exceptions are detected when an instruction executes out-of-order and saved in the ROB. Since instructions commit in-order from the ROB, exceptions can still be taken in program order by not actually taking an exception until the corresponding instruction is at the head of the ROB, about to commit.

Problem 3.B

Consider an out-of-order processor with register renaming using a unified physical register file. A new physical register is allocated for each instruction's destination register in the decode stage, but since physical registers are a finite

resource, they must be deallocated at some point in time. Carefully explain when it is safe to deallocate a physical register.

The physical register can be freed when the *next* writer of the same *architectural* register commits. At that point, it is guaranteed that no instructions remaining in the pipeline need to read the old physical register.

Problem 4 : important features

For the following snippets of code, select the single architectural feature that will *most* improve the performance of the code. Explain your choice, including description of why the other features will not improve performance as much and your assumptions about the machine design. The features you have to choose from are: out-of-order issue with renaming, branch prediction, and superscalar execution. Loads are marked whether they hit or miss in the cache.

Problem 4.A

ADD.D F0, F1, F8
ADD.D F2, F3, F8
ADD.D F4, F5, F8
ADD.D F6, F7, F8

Out-of-Order Issue with Renaming
Branch Prediction
Superscalar (√)

Superscalar because the instructions have no dependencies so they could be issued in parallel (superscalar could deliver speedup).
The other two techniques will waste hardware because they will offer no speedup. Out-of-Order with renaming will not help because there are no dependencies. Branch prediction is useless since there are no branches.

Problem 4.B

```
loop: ADD R3 R4 R0
      LD R4, 8(R4)        # cache hit
      BNEQZ R4, LOOP
```

Out-of-Order Issue with Renaming
Branch Prediction (√)
Superscalar

Branch prediction is necessary with a tight loop to prevent bubbles in the pipeline.

Superscalars will be limited not only by the branches, but also the WAR and RAW hazards. They will limit how much ILP it can achieve.

Out-of-order with renaming will be limited by the branches. Renaming will take care of the WAR hazard, but the RAW hazard will still limit how much improvement is possible.

Problem 4.C.

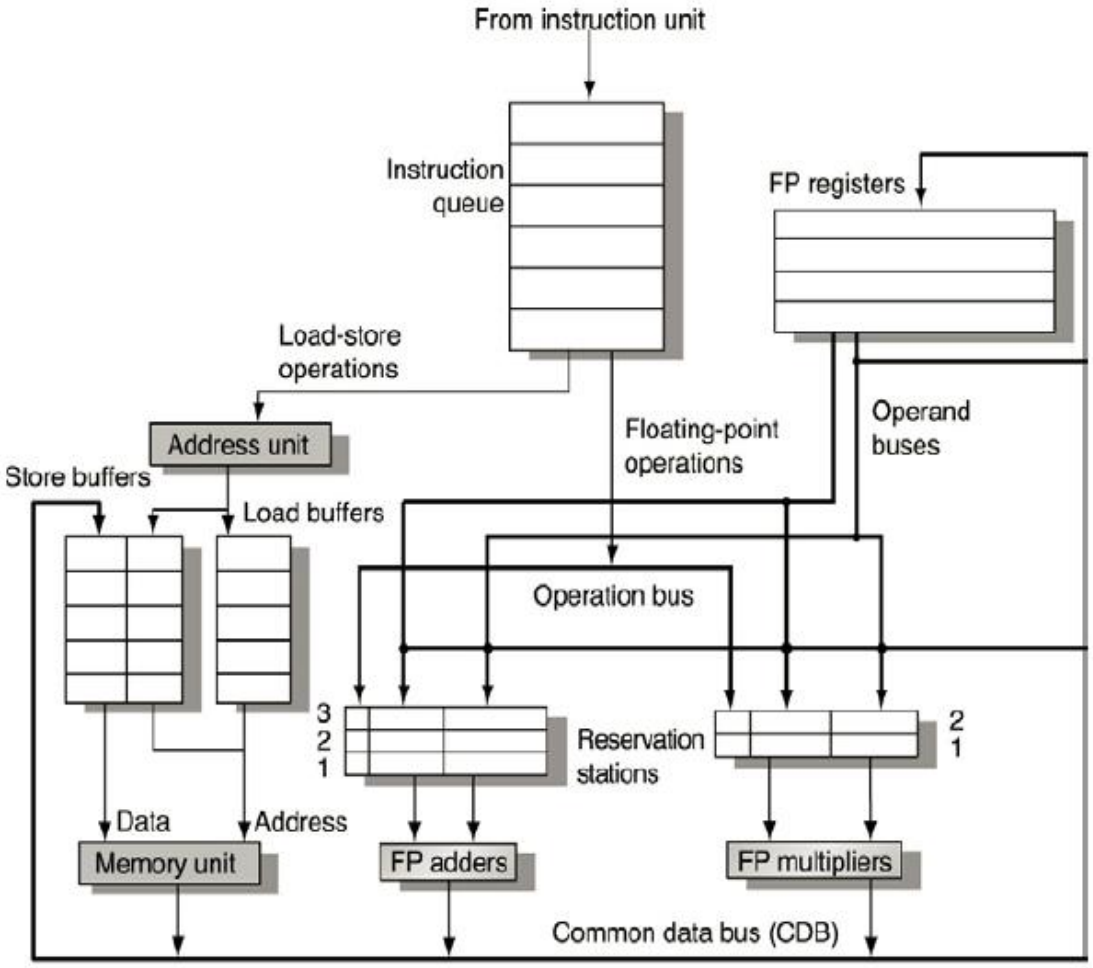
LD R1 0(R2) # cache miss
ADD R2 R1 R1
LD R1 0(R3) # cache hit
LD R3 0(R4) # cache hit
ADD R3 R1 R3
ADD R1 R2 R3

Out-of-Order Issue with Renaming (√)
Branch Prediction
Superscalar

Out-of-order with renaming will let the third, fourth, and fifth instruction run while the cache miss is being handled. When the miss completes it only needs to do the second and sixth instruction.
Branch prediction won't help with getting around the cache miss and there are enough hazards to limit a superscalar.

Problem 5. Tomasulo's 算法

下面考虑使用 Tomasulo's 算法来实现动态调度。处理器的数据通路如下图所示。处理器中各个功能单元： load/store 单元、FP ADD 单元、FP Mult 单元、Integer/Branch 操作单元都只有一个； 各个功能单元（function units）的工作延迟为： load/store 单元 2 周期，FP ADD 单元 3 周期， FP Mult 单元 5 周期，Integer/Branch 单元 1 周期。所有功能单元都是流水化的。如果一个操作数在某一个周期写回，依赖于该操作数的指令可以在下一周期开始执行。



A. **填充下面的表格，**显示在 Tomasulo's 算法动态调度下，各条指令在各个阶段的周期数。假设从第 0 周期开始。

在这个小题里，我们假设所有功能单元前端对应的保留站（reservation station）都只有一个表项。因为进入保留站的指令的结果操作数是用保留站号来标记的，所以当保留站中有一条指令，如果该指令已经开始执行，不能将该保留站清除并重新分配给新发射的指令。例如：load/store 指令会因为缺少保留站引发 structural hazard（结构冲突）导致的停顿。引起了下表中第二条 load 指令的发射必须延迟到第 4 周期才能开始。

Solution I	**Issue**	**Execute**	**WB**
Code	**Issue**	**Execute**	**WB**
L.D F2, 0(R1)	0	1-2	3
MUL.D F4, F2, F0	1	4-8	9
L.D F6, 0(R2)	4	5-6	7
ADD.D F6, F4, F6	5	10-12	13
S.D F6, 0(R2)	8	14-15	
DADDUI R1, R1, #8	9	10	11
DADDIU R2, R2, #-8	12	13	14
BGT R1, #800	15	16	

B. 接下来，假设数据通路中有一个 4 个 entry(表项）的 ROB（Reorder Buffer: 重排序缓冲器），如下图所示。当 问题 A 中的 ADD.D 指令已经准备好提交（但未提交）时，填写 ROB 中这 4 个 entry 的状态。表中 Complete？这一栏表示该指令是否已经执行。

Entry	Instruction	Destination	Complete?
1	S.D F6, 0(R2)	M[R2 + 0]	NO
2	DADDUI R1, R1, #8	R1	YES
3	DADDIU R2, R2, #-8	R2	YES
4	ADD.D F6, F4, F6	F6	YES

C. 考虑下面这段循环代码：

```
1.      LOOP: LD F2, 0(R1)
2.            LD F4, 0(R2)
3.            MULT F6, F2, F4
4.            SUBD F4, F6, F10
5.            ADDD F8, F8, F4
6.            DADDIU R1, R1, #-8
7.            DADDIU R2, R2, #-8
8.            BNEZ R1, R4 LOOP
```

填充下面的表格，显示在 Tomasulo's 算法动态调度下，各条指令在各个阶段的周期数。假设从第 0 周期开始。但是在本小题，与上一小题不同的是：我们假设所有功能单元的前端对应的保留站（reservation stations)，每个保留站有两个表项（entry)：

Instruction	Issue	Execute	WB
LD F2, 0(R1)	0	1-2	3
LD F4, 0(R2)	1	2-3	4
MULT F6, F2, F4	2	5-9	10
SUBD F4, F6, F10	3	11-13	14
ADDD F8, F8, F4	4	15-17	18
DADDIU R1, R1, #-8	5	6	7
DADDIU R2, R2, #-8	6	7	8
BNEZ R1, R4 LOOP	8	9	

问题 C 给出的指令中，哪一条指令的结果不会被写回到对应的寄存器文件？ 为什么？

Solutions；

LD F4, 0(R2)

SUBD 指令有 WAW 相关性，写回同一个寄存器

D. 将**问题** C 中的那段代码，静态调度到一个两路超标量机（two way superscalar machine)上执行，在这个双发射的机器上，每一周期最多发射两条指令，其中一条必须是浮点运算指令，另外一条是访存指令、整数运算指令或者转移指令。在给定的这个机器上，为这段代码（仅仅一个循环体，不进行循环展开）设计你认为最好的指令调度。

Only the LD or DADDIU instructions can be paired with any of the floating point instructions. Concurrency between remaining pairs of instructions is limited by RAW and issue constraints.

E. 解释一下 Tomasulo 算法是怎么避免 WAW 冲突的？ 用以下代码来作为示例进行解释。

```
ADD.D    F0, F2, F4
...
...
SUB.D    F0, F6, F8
```

F. 解释一下 Tomasulo 算法是怎么避免 WAR 冲突的？ 用以下代码来作为示例进行解释。

```
ADD.D    F0, F2, F4
...
...
MUL.D    F2, F6, F8
```