COMP 4611: Design and Analysis of Computer Architectures
Homework #2 Solutions

Q1 [10 points] Pipeline Hazards

The logic of a program naturally introduces dependences among instructions, including data dependence (true dependence), name dependence (anti-dependence, output dependence), and control dependence (when the decision of whether to execute an instruction is determined by the outcome of a branch). In some pipelines a specific dependence may result in a hazard that prevents the dependent instruction from moving forward in the pipeline without stalls. In some other pipelines, the same dependence may not result in a hazard. Consider the following MIPS assembly code.

```
1  LD     R1, 45(R2)
2  DADD   R7, R1, R5
3  DSUB   R8, R1, R6
4  OR     R1, R5, R1
5  BNEZ   R7, branch_target
6  DADD   R10, R10, R5
7  XOR    R2, R3, R4
```

Instruction 4 (OR R1, R5, R1) has a true dependence and an output dependence on instruction 1 (LD R1, 45(R2)). However, in the basic 5-stage MIPS pipeline, these dependences do not result in a hazard because instruction 4 is far enough from instruction 1, and the basic 5-stage MIPS pipeline writes the result back to register file in the last (5^{th}) stage. Similarly, instruction 7 (XOR R2, R3, R4) has a control dependence on instruction 5 (BNEZ R7, branch_target), but this dependence would not introduce a hazard in a 5-stage pipeline if the outcome and target address of a branch are computed in the ID (2^{nd}) stage.

Identify all dependences by type in the above code segment; list the two instructions involved; identify which instruction is dependent; and, if there is one, name the storage location involved.

	Dependence type	Independent instruction	Dependent instruction	Storage location
1 (example)	Data	1(LD)	2(DADD)	R1
2	Data	1(LD)	3(DSUB)	R1
3	Data	1(LD)	4(OR)	R1
4	Name	1(LD)	4(OR)	R1
5	Name	1(LD)	7(XOR)	R2
6	Name	2(DADD)	4(OR)	R1
7	Data	2(DADD)	5(BNEZ)	R7
8	Name	3(DSUB)	4(OR)	R1
9	Control	5(BNEZ)	6(DADD)	
10	Control	5(BNEZ)	7(XOR)	

Beyond the question (Informational, not used for grading):

We can further discern the dependences as shown in the table below.

	Dependence type	Independent instruction	Dependent instruction	Storage location
1 (example)	Data	1(LD)	2(DADD)	R1
2	True	1(LD)	3(DSUB)	R1
3	True	1(LD)	4(OR)	R1
4	Output	1(LD)	4(OR)	R1
5	Anti-dependence	1(LD)	7(XOR)	R2
6	Anti-dependence	2(DADD)	4(OR)	R1
7	True	2(DADD)	5(BNEZ)	R7
8	Anti-dependence	3(DSUB)	4(OR)	R1
9	Control	5(BNEZ)	6(DADD)	
10	Control	5(BNEZ)	7(XOR)	

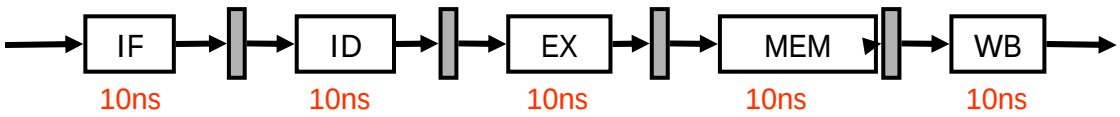
Q2. [10 Points] Pipeline performance

This question assumes single-issue pipelines -- at most one instruction enters the pipeline per clock cycle.

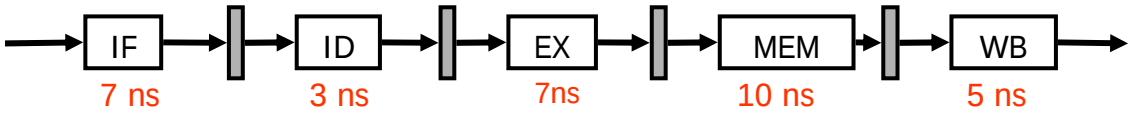
i. (4 points) What is the best CPI we can get with the 5-stage pipeline if in an ideal situation?

1

ii. (6 points) Suppose we have an old implementation of the 5-stage basic MIPS pipeline. In the old implementation, each instruction spends 10ns in each of the five pipeline stages. Now we have hired a new engineer who is very good at circuit design. He tells us that he can improve the circuit in the five stages so that the time it takes for an instruction to complete its operation in the 5 stages becomes 7ns, 3ns, 7ns, 10ns, 5ns, respectively, as shown in the figure below.



Old pipeline



New pipeline proposed by the new engineer

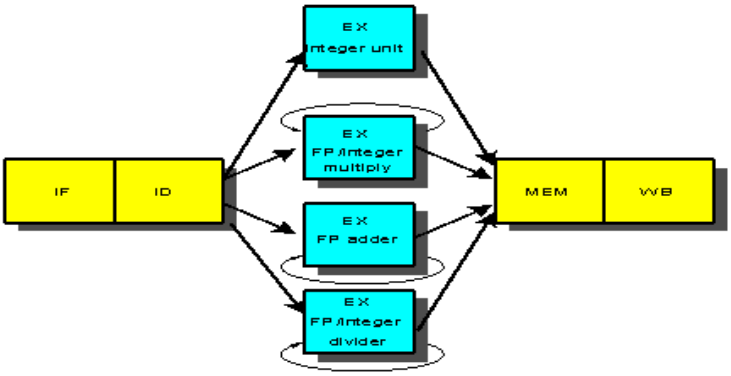
Can this proposed new pipeline enhance the performance? Why?

No. The performance of a pipeline depends on the slowest pipeline stage which takes the longest time. Since the longest stage in the proposed pipeline is still 10ns, which is the same as before the improvement, the performance of the pipeline is not enhanced.

Q3 [15 points] Pipelining without/with Forwarding

A multiple-execution pipelined processor with multiple execution functional units is shown below. In the execution stage, the Integer Unit takes 1 cycle, the FP/Integer Multiply Unit is pipelined and takes 4 cycles, the FP Adder Unit is pipelined and takes 3 cycles and the FP/Integer Divider is NOT pipelined and takes 5 cycles. Assume the data memory (data cache) can service one read or write per clock cycle. The instruction memory (instruction cache) works independently of the data memory. Assume the register file supports first half cycle write and second half cycle read. Note this is a single-issue and in-order issue pipeline.

In this question, you can use short forms like F, D, E, M1, M2, M3, M4, D1, D5, M, W to represent IF, ID, EX, Multiply stage 1, Multiply stage 2, Multiply stage 3, Multiply stage 4, Divide cycle 1, Divide cycle 5, MEM, WB, and so on.



i. (7 points) Fill in the pipeline timing chart for the code segment WITHOUT forwarding.

Instruction	Clock Cycles																								
	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
LD F2, 1000(R0) (example)	F	D	E	M	W																				
LD F3, 1004(R0)		F	D	E	M	W																			
DIVF F0, F2, F3			F	D	S	S	D1	D2	D3	D4	D5	M	W												
ADDF F1, F0,45				F	S	S	D	S	S	S	S	S	S	A1	A2	A3	M	W							
SUBF F2, F2, F3							F	S	S	S	S	S	S	D	A1	A2	A3	M	W						
LD R4, 1008(R0)														F	D	E	S	S	M	W					
MULTF F3, F1, F2															F	D	S	S	S	M1	M2	M3	M4	M	W

ii. (8 points) Fill in the pipeline timing chart for the code segment WITH forwarding. The forwarding logic can forward data from any execution functional unit's final output and the MEM/WB latch to any execution functional unit's initial input.

Instruction	Clock Cycles																								
	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
LD F2, 1000(R0) (example)	F	D	E	M	W																				
LD F3, 1004(R0)		F	D	E	M	W																			
DIVF F0, F2, F3			F	D	S	D1	D2	D3	D4	D5	M	W													
ADDF F1, F0,45				F	S	D	S	S	S	S	A1	A2	A3	M	W										
SUBF F2, F2, F3						F	S	S	S	S	D	A1	A2	A3	M	W									
LD R4, 1008(R0)											F	D	E	S	S	M	W								
MULTF F3, F1, F2												F	D	S	M1	M2	M3	M4	M	W					

Q4 [20 Points] Branch Prediction

Suppose we have a program with the following sequence of statements. It has three branches as indicated by B1, B2 and B3.

…
if (a<b) then a=2*a; # branch B1
if (c>b) then c=c-b; # branch B2
if (a>c) then a=a-b; # branch B3
…

The instruction sequence corresponding to the above statements is shown in Fig. 1 in assembly language. In Fig. 1, register R1 is used for the variable a, R2 for b and R3 for c. R4 is a register to store temporary results. We maintain a (m,n) predictor for each branch and the predictor for the branch B3 is illustrated in the following table (Fig. 2).

```
      …
S1:   SUB  R4, R1, R2  ;   R4=R1-R2
B1:   BGE  R4, S2      ;   if R4≥0, then branch to S2 (B1 branch)
      ADD  R1, R1, R1  ;   R1=R1+R1
S2:   SUB  R4, R3, R2  ;   R4=R3-R2
B2:   BLE  R4, S3      ;   if R4≤0, then branch to S3 (B2 branch)
      SUB  R3, R3, R2  ;   R3=R3-R2
S3:   SUB  R4, R1, R3  ;   R4=R1-R3
B3:   BLE  R4, S4      ;   if R4≤0, then branch to S4 (B3 branch)
      SUB  R1, R1, R2  ;   R1=R1-R2
S4:   …
```

Fig. 1: The sequence of instructions

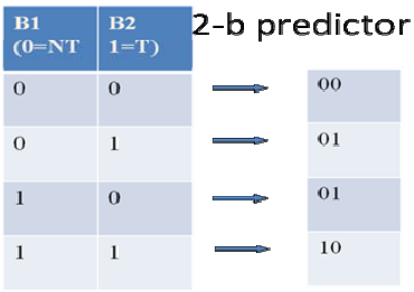


Fig. 2: The (m,n) predictor for branch B3

i. (3 points) For the (m, n) predictor, what is the parameter m and what is the parameter n?

m=2 and n=2.

ii. (6 points) Suppose at certain time instance, the variables a=26, b=50 and c=46. The program counter (PC) points the first instruction at label S1 (SUB R4, R1, R2). Suppose the state of predictor of B3 at the time instance is shown in Fig. 2. According to this predictor, what prediction will be made for the branch B3 (TAKEN or NOT TAKEN)? Explain the reason.

a=26 and b=50, so B1 is "NOT TAKEN". c=46 and b=50, so B2 is "TAKEN" According to the outcomes of branches B1 and B2, the predictor indexed by NT/T is selected. Since the so-far state is "01", we will made the prediction of "NOT TAKEN"

iii. (6 points) Follow the conditions of the question b. When the program has just finished the execution of the branch B3 (i.e., PC becomes more than B3), what will be the state of the predictor of B3?

When PC=B3, we have a=52 and c=46. Therefore B3 is "NOT TAKEN". The state of the corresponding 2-bit predictor (the '01' entry) will be changed from 01 to 00.

B1	B2	Predictor
0	0	00
0	1	00
1	0	01
1	1	10

iv. (5 points) Suppose we can use up to 10000 bits for dynamic branch prediction using this (m,n) predictor scheme. How many entries can we hold in the cache at most? Assume the number of entries is a power of 2, and each entry corresponds to a different instruction address. (Hint: m and n are determined in question a)

M=2 and n=2. So each entry needs 2^2*2=8bits. We have 10000 bits and therefore we can hold (10000/8)=1250 entries. We support only the number of power of 2 entries, 1250 is grounded to 1024 entries.

Instruction examples: In some questions, you may need to read or write MIPS instructions in assembly language. Below are some examples of instructions in assembly code:

Load a word from memory to r1:	**LW r1, 100(r2)**
Load 64 bits from memory to F0 (double precision):	**LD F0,100(r2)**
Store a the word in r3 to memory:	**SW r3, 100(r4)**
Add r6 and r7 and assign the result to r5:	**ADD r5, r6, r7**
Add r6 and an immediate, and assign the result to r5:	**ADD r5, r6, 200**
Add F1 and F2 and assign the result to F3 (single-precision):	**ADD.S F3, F1, F2**
Branch to address indicated by Label9 if r8 is zero:	**BEQZ r8, Label9**

You can use ';' to start comments in a line, e.g.,

```
LW r1, 100(r2); Here is the comment within one line
```

Q5 [20 points] Code scheduling

The following code calculates the floating-point expression E = A + B + C * D, where the memory addresses of A, B, C, D, and E are stored in R1, R2, R3, R4, and R5, respectively:

```
L.S     F0, 0(R1)
L.S     F1, 0(R2)
ADD.S   F0, F0, F1
L.S     F2, 0(R3)
L.S     F3, 0(R4)
MUL.S   F2, F2, F3
ADD.S   F0, F0, F2
S.S     F0, 0(R5)
```

X.S means performing operation X on single-precision data. For example, L.S means loading a single-precision number.

a. [8 points] Suppose we execute the above code segment on an in-order pipelined machine. Show the stalls we need to insert between instructions, and calculate the number of cycles this code sequence would take to execute (i.e., the number of cycles between the issue of the first load instruction and the issue of the final store, inclusive). For simplicity, you can think of the "issue" cycle (or the cycle in which an instruction is issued) as the cycle immediately before an instruction enters its execution functional unit for execution.

Consider the memory load/store unit as a functional unit in parallel with the FP ALU units. Assume the following latencies:

Load → FP ALU:	2 cycles
FP multiplication → FP ALU:	4 cycles
FP multiplication → Store:	3 cycles
FP addition → FP ALU:	2cycles
FP addition → Store:	2cycles

As an example, if a load instruction (e.g., L.S) is issued in cycle 1, an ALU instruction that uses the load instruction's result can be issued in cycle 4 and proceed in the pipeline without stalls. Assume that all functional units are fully pipelined and ignore any write back conflicts on the register file.

```
1       L.S  F0, 0(R1)
2       L.S  F1, 0(R2)
3       Stall       ; Load → FP ALU:          2 cycles
4       Stall
5       ADD.S   F0, F0, F1
6       L.S  F2, 0(R3)
7       L.S  F3, 0(R4)
8       Stall       ; Load → FP ALU:          2 cycles
9       Stall
10      MUL.S   F2, F2, F3
11      Stall       ; FP multiplication → FP ALU: 4 cycles
12      Stall
13      Stall
14      Stall
15      ADD.S   F0, F0, F2
16      Stall       ; FP addition → Store:       2cycles
17      Stall
18      S.S  F0, 0(R5)
```

Totally, 18 cycles.

b. [12 points] Reorder the instructions in the code sequence to minimize the execution time. Show the new instruction sequence, and indicate the position and number of stalls we need to insert between instructions.

```
1       L.S  F2, 0(R3)
2       L.S  F3, 0(R4)
3       L.S  F0, 0(R1)
```

```
4       L.S F1, 0(R2)
5       MUL.S   F2, F2, F3
6       Stall       ; FP multiplication → FP ALU: 4 cycles
7       ADD.S   F0, F0, F1
8       Stall
9       Stall
10      ADD.S   F0, F0, F2
11      Stall       ;FP addition → Store:     2cycles
12      Stall
13      S.S F0, 0(R5)
```

Totally, 13 cycles.

Q6 [15 points] VLIW machines

The program we will use for this problem is listed below (In all questions, you should assume that arrays **A**, **B** and **C** do not overlap in memory).

C code

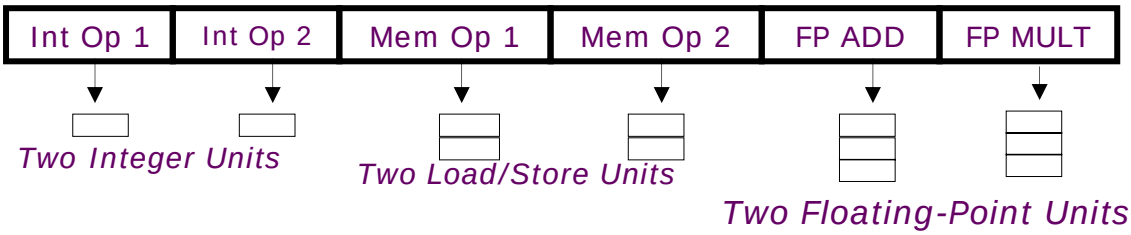
```
for (i=80; i>0; i--) {
    A[i] = A[i] * B[i];
    C[i] = C[i] + A[i];
}
```

In this problem, we will deal with the code sample on a VLIW machine. Because the software keeps track of the dependences among instructions and schedules them in the long instruction words, our VLIW machine does not use interlocks for hazard prevention. The machine does not use forwarding, and the result of an operation is written to the register file immediately after it has gone through the corresponding execution functional unit. Our machine has six execution units:

- 2 integer ALU units, also used for branch operations. The latency between an integer ALU operation and any other operation is 1 cycle.
- 2 memory units, fully pipelined. Each unit can perform either a store or a load. The latency between a memory operation and any other operation is 2 cycles.
- 2 FP units, fully pipelined. One unit can perform **`fadd`** operations, the other **`fmul`** operations. The latency between an FP ALU operation and any other operation is 3 cycles.

As an example, if an integer ALU operation enters the integer ALU unit in cycle 1, another operation using the integer ALU operation's result can enter its execution functional unit in cycle 3 and proceed in the pipeline without stalls.

Below is a diagram of our VLIW machine:



The loop in the program can be translated to the following operations. Suppose r1, r2 and r3 initially contain the addresses of A[80], B[80], and C[80], and r4 initially contains the number 80.

```
loop:    1.  ld f1, 0(r1)       ; f1 = A[i]
         2.  ld f2, 0(r2)       ; f2 = B[i]
         3.  fmul f4, f2, f1    ; f4 = f1 * f2
         4.  st f4, 0(r1)       ; A[i] = f4
         5.  ld f3, 0(r3)       ; f3 = C[i]
         6.  fadd f5, f4, f3    ; f5 = f4 + f3
         7.  st f5, 0(r3)       ; C[i] = f5
         8.  add r1, r1, -4
         9.  add r2, r2, -4
        10.  add r3, r3, -4
        11.  add r4, r4, -1     ; i--
        12.  bnez r4, loop      ; loop
```

i. [7 points] Table 6.a shows our program rewritten for our VLIW machine, with some operations missing (operations `2, 6` and `7`). We have adjusted and rearranged the operations to let them execute as soon as they possibly can, while ensuring program correctness. Assume the VLIW machine's processor reads operands for the operations before issuing them into the execution functional unit. Hence, combining operations such as 1 and 8 in one instruction word is possible. Please fill in missing operations in Table 6.a. (Note, you may not need all the rows)

ALU1	ALU2	MU1	MU2	FADD	FMUL
`add r1, r1, -4`	`add r2, r2, -4`	`ld f1, 0(r1)`	ld f2, 0(r2)		
`add r3, r3, -4`	`add r4, r4, -1`	`ld f3, 0(r3)`			
					`fmul f4, f2, f1`
			`st f4, 4(r1)`	fadd f5, f4, f3	
	`bnez r4, loop`	st f5, 4(r3)			

Table 6.a: VLIW Program

ii. [8 points] Assume the VLIW processor has 32 FP registers (f0 – f31). If we unrolled the loop once (combine two iterations to be one larger iteration), would that give us better performance? How shall we (or the compiler) write the VLIW instructions to accomplish the best performance? Write the instructions in Table 6.b.

ALU1	ALU2	MU1	MU2	FADD	FMUL
add r4, r4, -2	**ADD R3, R3, -8**	**ld f1, 0(r1)** **f1 ← A[i]**	LD F2, 0(R2) **f2 ← B[i]**		
ADD R1, R1, -8	**ADD R2, R2, -8**	**ld f9, -4(r1)** **f9 ← A[i-1]**	LD F10, -4(R2) **F10 ← B[i-1]**		
		LD F3, 8(R3) **f3 ← C[i]**	LD F11, 4(R3) **f11 ← C[i-1]**		
					fmul f4, f2, f1 **f4 ← B[i]* A[i]**
					FMUL F12, F10, F9 **f12←B[i-1]*A[i-1]**
		ST F4, 8(R1) **A[i]←B[i]* A[i]**		FADD F5, F4, F3 **f5←B[i]*A[i]+C[i]**	
		ST F12, 4(R1) **A[i-1]←B[i-1]*A[i-1]**		FADD F13, F12, F11 **f13←B[i-1]*A[i-1]+C[i-1]**	
		ST F5, 8(R3) **C[i]←B[i]*A[i]+C[i]**			
	BNEZ R4, loop	ST F13, 4(R3) **C[i-1]←B[i-1]*A[i-1]+C[i-1]**			

Table 6.b: VLIW Program

Q7 [10 points] Pipeline performance with stalls

Assume the following MIPS instruction mix:

Type	Frequency	
Arith/Logic	50%	
Load	30%	of which 25% are followed immediately by an ALU instruction using the loaded value
Store	5%	
Branch	15%	of which 30% are taken

What is the resulting CPI for the pipelined MIPS with forwarding and branch address calculation in ID stage when using a predict branch not-taken scheme?

CPI = Ideal CPI + Pipeline stall clock cycles per instruction
= 1 + stalls by loads + stalls by branches
= 1 + .3 x .25 x 1 + .15 x .30 x 1
= 1 + .075 + .045
= 1.12