ECE4700J Computer Architecture

Summer 2022

HW #3

Due: **2:59am (Beijing Time) June 22nd, 2022**

*Please submit a single **PDF** file on Canvas*

Q1 (10%):

1) (5%) How many bypass paths are there in a 5-stage (F, D, X, M, W) pipeline that is 4 instructions wide? How many point-to-point wires and multiplexors (and what size muxes) does this bypass network requires, compared to a processor without bypassing?

2) (5%) What if you use k-clustering techniques such as group ALUs in 2 clusters? Repeat the above questions.

Sample Solution:

1) There are 4 × 2 × 4 × 2 = 64 paths in total. There are 64 point-to-point wires and 8 9-to-1 MUXs.
2) There are 32 bypass paths, thus 32 point-to-point wires and 8 5-to-1 MUXs.

Q2 (20%):

1) (10%) Identify the write-read, write-write, and read-write dependencies in the instruction sequence below by entering each line pair with a dependency in the correct column of the table to the right. For example, if L1 and L4 had a write-write dependency (which they don't), you would enter L1-L4 in the column labeled "write-write"

```
L1: R1 = 100
L2: R1 = R2 + R4
L3: R2 = R4 - 25
L4: R4 = R1 + R3
L5: R1 = R1 + 30
```

write-read	write-write	read-write
*L2 - L4	*L1 - L2	L2 - L3
*L2 - L5	*L2 - L5	L2 - L4
L1 - L4	L1 - L5	L3 - L4

L1 – L5		L4 – L5

2) (10%) Rename the registers from 1) to prevent dependency problems. You can assume there are free registers (R5 – R10). Show your work step by step.

Sample Solution:

L1: R5= 100
L2: R6 = R2 + R4
L3: R7 = R4 - 25
L4: R8 = R6 + R3
L5: R9 = R6 + 30

Q3 (10%): Read about VLIW and list at least two limitations regarding VLIW. You can feel free to use internet for this question.

Sample Solution:

- Complex compilers are required which are hard to design.
- Increased program code size.
- Larger memory bandwidth and register-file bandwidth.
- Unscheduled events, for example a cache miss could lead to a stall which will stall the entire processor.
- In case of un-filled opcodes in a VLIW, there is waste of memory space and instruction bandwidth.
- …

Q4 (40%): Given the following instruction sequence, and we also assume the following conditions:

a. The pipeline is a two-way superscalar.
b. The Memory stage takes two cycles to finish.
c. Loads have two-cycle load-to-use penalty (three cycle total execution latency).
d. Multiplication takes three cycles to finish.
e. All other instructions have single-cycle execution latency

Instruction	C0	C1	C2	C3	C4	C5	C6	….	
`ld [r1] → r3`									
`mult r3*r2 → r4`									
`add r2+r5 → r4`									
`or r4^r6 → r7`									
`sub r7-3 → r8`									
`ld [r9] → r7`									
…									

To enable the OoO execution, you can use a map table, ready table, reorder buffer and an issue queue. The following is the initial state.

Map Table

r1	p8
r2	p7
r3	p9
r4	p2
r5	p5
r6	p1
r7	p4
r8	p3
r9	p6

Ready Table

p1	Yes
p2	Yes
p3	Yes
p4	Yes
p5	Yes
p6	Yes
p7	Yes
p8	Yes

p9	Yes
p10	
p11	
p12	
p13	
p14	
p15	
p16	
p17	
p18	

Reorder Buffer

Instruction	To Free?	Done?
….		
…		

Issue Queue

Instruction	Src1	R?	Src2	R?	Dest	Bday

Question: How would this execution occur cycle-by-cycle? Show your work in details.

Sample Solution:

The final status (Notice that the implementation can be slightly different depending on your assumptions, the following shows one possible example).

Instruction	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
`ld [r1] → r3`	F	Di	I	RR	X	M1	M2	W	C			
`mult r3*r2 → r4`	F	Di				I	RR	X	X	X	W	C
`add r2+r5 → r4`		F	Di	I	RR	X	W					C
`or r4^r6 → r7`		F	Di		I	RR	X	W				C
`sub r7-3 → r8`			F	Di		I	RR	X	W			C
`ld [r9] → r7`			F	Di	I	RR	X	M1	M2	W		C
…												

Map Table

r1	p8
r2	p7
r3	p10
r4	p12
r5	p5
r6	p1
r7	p15
r8	p14
r9	p6

Ready Table

p1	Yes
p2	—
p3	—
p4	—
p5	Yes
p6	Yes
p7	Yes
p8	Yes
p9	—
p10	Yes
p11	—
p12	Yes
p13	—
p14	Yes
p15	No
p16	
p17	
p18	

Reorder Buffer

Instruction	To Free?	Done?
~~ld~~	~~p9~~	~~yes~~
~~mult~~	~~p2~~	~~yes~~
~~add~~	~~p11~~	~~yes~~
~~or~~	~~p4~~	~~yes~~
~~sub~~	~~p3~~	~~yes~~
~~ld~~	~~p13~~	~~yes~~

Q5 (20%):

1) (10%) Suppose that we want loads and stores to execute out-of-order with respect to each other. Under what circumstances in the code below can we execute instruction 5 before executing any others?

```
1  add x1, x1, x2
2  sw  x5, (x2)
3  lw  x6, (x8)
4  sw  x5, (x6)
5  lw  x8, (x3)
6  add x8, x8, x8
```

Sample Solution:

We can only execute instruction 5 before the others if the address it is loading from does not overlap the addresses being stored to in instructions 2 and 4. So $|x2 - x3| \geq 4$ and $|x6 - x3| \geq 4$. Another case is when x2 is not equal to x3, and x3 is not equal to x6.

2) (10%) Under what circumstances can we execute instruction 4 in the code above before executing any others? Is it possible? Please explain your answer.

Sample Solution:

Instruction 4 has a RAW hazard with instruction 3 because it depends on the value of register x6 that is being loaded by instruction 3. Therefore, it cannot execute first. It does not have any dependencies on any other instruction, since stores do not change architectural state until they are committed, so it can execute as soon as instruction 3 completes.

Optional Question:

OQ1 (10%): In an in-order single scalar 5-stage pipeline, consider a branch with the following behavior:

```
T, T, T, T, T, N, N, N, N, N, N, N, N, T, T, T, T, T, T, T, T, T,
T, N, N, N, N, N, T, T, T, T, T, T
```

N stands for not-taken and T stands for taken. You need consider only static, n-bit branch predictors as discussed in VE370, where m and n could be either 1 or 2. Your explanations can be based on inspection of the branch behavior; you do not have to work out actual branch misprediction rates in any case.

Q: Would you suggest using a 2-bit predictor for this branch? If yes, why? If not, which predictor should be used instead? Show your work.

Sample Solution:
Using a 2-bit predictor would lead to 2 mispredicts each time the branch behavior changes. It would be better to use a 1-bit predictor, it would lead to 4 mispredicts, whereas a 2-bit predictor would lead to 8 mispredicts.