Lecture 23: Board Notes: Introduction to Parallel Processing

Part A:

Consider a processor that does register renaming.

- Each instruction must spend at least 1 CC in a reservation station
- ALU operations take 1 CC to execute.
  - There are an unlimited number of functional units.
- If an instruction in a RS is waiting for data produced by a previously issued instruction, it will obtain that data during the previously issued instruction's WB stage – and can execute *in the next CC.*
  - i.e. if instruction *j* enters WB in cycle 7, and instruction *j+4* is waiting on data from instruction *j*, instruction *j+4*'s RS will be updated in cycle 7. Instruction *j+4* can execute in cycle 8
- Only 1 instruction is fetched and decoded during each clock cycle.
- Assume RS are unlimited.
- There are unlimited common data bus resources. Therefore there are no structural hazard stalls when instructions need to write back.
- 2 instructions may commit in each CC.
- Multiply instructions take 4 CCs to execute, Adds take 1 CC to execute.

Fill in the pipe trace for the instruction sequence shown on the next page.
(F) Fetch, **(D)** Decode, **(RS)** Reservation Station, **(E)** Execute, **(W)** Write Back, **(C)** Commit

	PART A																		
	Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
A	**Add r1 ,r1, r1**	**F**	**D**	**R**	**E**	**W**	**C**												
B	**Add r1, r1, r1**		**F**	**D**	**R**	**R**	**E**	**W**	**C**										
C	**Mul r1, r1, r1**			**F**	**D**	**R**	**R**	**R**	**E**	**E**	**E**	**E**	**W**	**C**					
D	**Sub r2, r2, r2**				**F**	**D**	**R**	**E**	**W**	**C**	**C**	**C**	**C**	**C**					
E	**Add r1, r2, r2**					**F**	**D**	**R**	**R**	**E**	**W**	**C**	**C**	**C**	**C**				
F	**Mul r2, r3, r3**						**F**	**D**	**R**	**E**	**E**	**E**	**E**	**W**	**C**				
G	**Add r1, r1, r1**							**F**	**D**	**R**	**R**	**E**	**W**	**C**	**C**	**C**			

Part B: Example 1:

Assume we want to split up a problem to run on 1024 processors instead of 1. However, only half of the code is parallelizable. What speedup would we see from going from 1 processor to 1024?

$$\text{speedup}_{\text{overall}} = \frac{1}{(1 - F_{parallel}) + \frac{F_{parallel}}{Speedup_{parallel}}} = \frac{1}{(1 - 0.5) + \frac{0.5}{1024}} = 1.998!$$

If the fraction of code that is parallelizable increases from 0.5 to 0.99, speedup is still only 91, not 1024!

Part B: Example 2:

Assume that we have a given workload that involves:

- Sum of 10 scalars
- 10 x 10 matrix sum

Part A:

What is the speedup if we increase the number of processors dedicated to the problem to 10? To 100?

1 Processor:

Time	=	(10 + 100)	x	t_{add}	=	110 x t_{add}

- 10 scalar adds + 100 adds for each element in the matrix

10 Processors:

Time	=	10 x t_{add}	+	(100/10) x t_{add}	=	20 x t_{add}
Speedup	=	110 x t_{add}	/	20 x t_{add}	=	5.5
		(best uniprocessor)			=	55 % of the potential (5.5 / 10)

100 Processors:

Time	=	10 x t_{add}	+	(100/100) x t_{add}	=	11 x t_{add}
Speedup	=	110 x t_{add}	/	11 x t_{add}	=	10
		(best uniprocessor)			=	10 % of the potential (10 / 100)

This assumed that the load can be balanced across processors

Part B:

What is the speedup if the matrix size is now 100 x 100?

1 Processor:

Time	=	(10 + 10000)	x	t_{add}	=	10010 x t_{add}

- 10 scalar adds + 10000 adds for each element in the matrix

10 Processors:

Time	=	10 x t_{add}	+	(10000/10) x t_{add}	=	1010 x t_{add}
Speedup	=	10010 x t_{add}	/	1010 x t_{add}	=	9.9
		(best uniprocessor)			=	99 % of the potential (9.9 / 10)

100 Processors:

Time	=	10 x t_{add}	+	(10000/100) x t_{add}	=	110 x t_{add}
Speedup	=	10010 x t_{add}	/	110 x t_{add}	=	91
		(best uniprocessor)			=	91 % of the potential (91 / 100)

This assumes load balancing is possible; if problem is smaller, scalar parts dominates (not parallel)

Part C:

In this question, you're going to leverage techniques that you've learned so far in class to quantitatively see how a multi-core computer architecture might improve overall performance (i.e. decrease execution time). We'll keep the discussion pretty simple for now…

Given the above context, assume that we want to compare 2 designs – each with its own execution model:

- Design 1 is a single-core machine with a 4 GHz clock rate.
- Design 2 is a dual-core machine with a clock rate that is 20% slower.

Assume that we are interested in how long it will take to execute all of the instructions associated with 2 processes on each design.

You know the following:

- Process 1 requires 2.5 million MIPS instructions
- Process 2 requires 6 million MIPS instructions
- In the tables below, I've listed the number of CCs each instruction "class" requires. Note that the number of CCs per class differs from design-to-design. The percentage of each instruction class per process is also listed.

Instruction Type	% (Process 1)	% (Process 2)
ALU	45%	65%
Store	12%	5%
Load	22%	15%
Branch/Jump	21%	15%

Instruction Type	CCs on Design 1	CCs on Design 2
ALU	4	4
Store	4	5
Load	5	6
Branch/Jump	3	3

(Note difference in shaded boxes)

On Design 1, Process 1 will be executed first, there will be a context switch (where we update the register file with the data for Process 2, etc. that will take 100,000 CCs), and then Process 2 will run until completion. On Design 2, each process can be mapped to a different core so there is no context switch overhead.

What performance improvement do we get by executing the instructions for these two processes on the dual core machine?

Solution:

CPU Time – Design 1, Process 1:

= 2.5M Instructions **x** [(0.45)(4) + (0.12)(4) + (0.22)(5) + (0.21)(3)] CCs / Inst **x** 0.25×10^{-9}s / CC
= 0.002506 s

Overhead:

= 100,000 CCs **x** 0.25×10^{-9}s / CC
= 0.000025 s

CPU Time – Design 1, Process 2:

= 6M Instructions **x** [(0.65)(4) + (0.05)(4) + (0.15)(5) + (0.15)(3)] CCs / Inst **x** 0.25×10^{-9}s / CC
= 0.006 s

Total: ~0.0085 s

CPU Time – Design 2, Process 1:

= 2.5M Instructions **x** [(0.45)(4) + (0.12)(5) + (0.22)(6) + (0.21)(3)] CCs / Inst **x** 0.313×10^{-9}s / CC
= 0.0034 s

CPU Time – Design 2, Process 2:

= 6M Instructions **x** [(0.65)(4) + (0.05)(5) + (0.15)(6) + (0.15)(3)] CCs / Inst **x** 0.313×10^{-9}s / CC
= 0.007875 s

Total: ~0.007875 s (b/c the processes run in parallel)

Therefore, 0.0085 / 0.00785 ~ 1.08 (therefore Design 2 is about 8% faster)

Part D:

Question:

- Assume that you have a system that uses 10000 disks
- The MTTF is 1,200,000 hours
- The disks are used 24 hours a day
- If a disk fails, you replace it with one that has the same reliability characteristics
- How many disks fail per year?

Failed Disks: (10000 drives) x (8760 hours / drive) / (1,200,000 hours/ failure) = 73

Thus, the Annual Failure Rate is 0.73%

But if in a supercomputing system, what if an entire computation must halt to replace???