CHALMERS

DAT 105 (DIT051) Computer Architecture Exam – Solution Exam Date: Jan 19, 2019

ASSIGNMENT 1

1A)

$$T = IC \text{ x } (CPI_0 + MPKI \text{ x } MP/10^3) \text{ x } Tc$$

Execution times P1 $IC=2 \text{ x } 10^6$

- A: $CPI_0=0.5$, MPKI=10, MP=100ns/1ns=100, T_c=1ns
  $T_{A,P1}= 2 \text{ x } 10^6 \text{ x } (0.5 + 10\text{x}100/10^3) \text{ x } 1ns = 2 \text{ x } 10^6 \text{ x } (0.5+1) \text{ x } 1 \text{ ns} =$ **3 ms**

- B: $CPI_0=1$, MPKI=10, MP=100ns/0.83ns=120, T_c=0.83ns
  $T_{B,P1}= 2 \text{ x } 10^6 \text{ x } (1 + 10\text{x}120/10^3) \text{ x } 0.83ns = 2 \text{ x } 10^6 \text{ x } (1+1.2) \text{ x } 0.83 \text{ ns} =$ **3.6 ms**

- R: $CPI_0=1$, MPKI=1, MP=100ns/1 ns=100, T_c=1 ns.
  $T_{R,P1}= 2 \text{ x } 10^6 \text{ x } (1 + 1\text{x}100/10^3) \text{ x } 1 \text{ ns} = 2 \text{ x } 10^6 \text{ x } 1.1 \text{ x } 1 \text{ ns} =$ **2.2 ms**

Execution times P2 $IC=1 \text{ x } 10^6$

- A: $CPI_0=2$, MPKI=10, MP=100ns/1ns=100, T_c=1ns.
  $T_{A,P2}= 1 \text{ x } 10^6 \text{ x } (2 + 10\text{x}100/10^3) \text{ x } 1ns = 1 \text{ x } 10^6 \text{ x } (2+1) \text{ x } 1 \text{ ns} =$ **3 ms**

- B: $CPI_0=1.5$, MPKI=10, MP=100ns/0.83ns=120, T_c=0.83ns.
  $T_{B,P2}= 1 \text{ x } 10^6 \text{ x } (1.5 + 10\text{x}120/10^3) \text{ x } 0.83ns = 1 \text{ x } 10^6 \text{ x } (1.5+1.2) \text{ x } 0.83 \text{ ns} =$ **2.2 ms**

- R: $CPI_0=1$, MPKI=1, MP=100ns/1 ns=100, T_c=1 ns.
  $T_{R,P2}= 1 \text{ x } 10^6 \text{ x } (1 + 1\text{x}100/10^3) \text{ x } 1 \text{ ns} = 1 \text{ x } 10^6 \text{ x } 1.1 \text{ x } 1 \text{ ns} =$ **1.1 ms**

1B) $\mathbf{SP_X = T_R / T_X}$

- P1: SP_A= 2.2/3=0.73 , SP_B= 2.2/3.6=0.61
- P2: SP_A=1.1/3=0.37 , SP_B=1.1/2.2=0.5

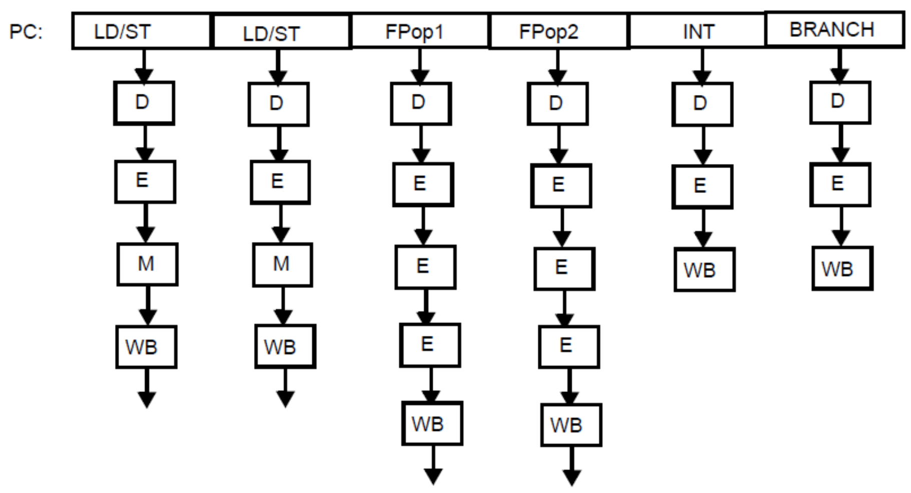
G-mean A=$\sqrt{0.73 \times 0.37} = 0.52$, G-mean B=$\sqrt{0.61 \times 0.5} = 0.55$,

Hence, B is on average slightly faster than A

1C) Speedup with enhanced functional unit:
SP = 1/(0.9 + 0.1/1000) = 1.1 (about 10% faster). This is a good example of not spending significant engineering efforts on a rare case.

1D) Arithmetic means are sensitive to outliers. So, if a computer outperforms another one for nine out of ten applications but underperforms significantly for one, arithmetic mean will suggest a low average performance.

ASSIGNMENT 2



Assumption: No forwarding

2A)

LOOP: I1: LD F1, 0(R1)
I2: ADD F4, F0, F2
I3: SD F4, 0(R1) → *wait for F4 value*
I4: SUBI R1, R1, #8
I5: BNE R1, R2, LOOP → *wait for R1 value*

Register read happens in the decode stage. Therefore, a consumer instruction can be in the decode stage when its input data is already available in the register file. That is one cycle after the write-back stage of the producer instruction. Hence:

- 4 cycles should be between I3 and I2
- 2 cycles should be between I5 and I4
- I3 reads R1 in the decode stage. I4 can modify R1 value after that. So, at the same cycle I4 can be in the WB stage. Hence, I4 can be scheduled two cycles earlier than I3 without causing a WAR hazard

Cycle	LD/ST	LD/ST	FPop1	FPop2	INT	BRANCH
1	LD F1,0(R1)		ADD F4,F0,F2			
2						
3						
4					SUBI R1,R1,#8	
5						
6	SD F4,0(R1)					
7						BNE R1, R2, LOOP

IPC= 5/7=0.7 Not much ILP is exploited.

2B)

Cycle	LD/ST	LD/ST	FPop1	FPop2	INT	BRANCH
1	LD F1,0(R1)	LD F6,-8(R1)	ADD F4,F0,F2	ADD F9,F5,F7		
2	LD F11,-16(R1)	LD F16,-24(R1)	ADD F14,F10,F12	ADD F19,F15,F17		
3	LD F21,-32(R1)	LD F26,-40(R1)	ADD F24,F20,F22	ADD F29,F25,F27		
4	LD F31,-48(R1)		ADD F34,F30,F32			
5						
6	SD F4,0(R1)	SD F9,-8(R1)				
7	SD F14,-16(R1)	SD F19,-24(R1)				
8	SD F24,-32(R1)	SD F29,-40(R1)			SUBI R1,R1,#56	
9	SD F34,-48(R1)					
10						BNE R1, R2, LOOP

IPC= 23/10=2.3

2C)

```
for (i = 0; i < N; i++)
    {A[i+2] = A[i] + 2;
     B[i] = A[i+2] + 1;}
```

This is translates into

```
Loop:       L.S F0, 0(R2)           ; –O1
            ADD.S F3, F0, F1        ; –O2
            ADD.S F4, F3, F1        ; –O3
            S.S. F3, 16(R2)         ; –O4
            S.S. F4, 0(R3)          ; –O5
            ADDI R2,R2, 8
            ADDI R3,R3, 8
            BNE R2,R4, Loop
```

See schedule in Table 3.25 on page 150 in the textbook. It contains the software pipelined schedule for the same code, assuming full forwarding. With the assumption of no forwarding, the distances between dependent instructions increases.

- Between a load operation (O1) and a dependent instruction (O2) we need 3 cycles
- Between a floating-point operation (O2, O3) and a dependent instruction (O3 & O4, O5) we need 4 cycles
- In this case, we also have a data dependency between iterations. O1 of iteration i+2 will read the value from O4 of iteration i as they are pointing to the same memory address. To respect this data dependency, O1(i+2) should be in memory stage when O4(i) has already passed that stage and written its value to memory (assuming a single cycle memory access time). Therefore, O1(i+2) should be scheduled one cycle after O4(i)

Hence, we have the following schedule assuming no forwarding:

	ITE1	**ITE2**	**ITE3**	**ITE4**	**ITE5**
INST1	O1				
INST2					
INST3					
INST4					
INST5	O2				
INST6		O1			
INST7					
INST8					
INST9					
INST10	O3,O4	O2			
INST11			O1		
INST12					
INST13					
INST14					
INST15	O5	O3,O4	O2		
INST16				O1	
INST17					
INST18					
INST19					
INST20		O5	O3,O4	O2	
INST21					O1
INST22					
INST23					
INST24					
INST25			O5	O3,O4	O2
INST26					
INST27					
INST28					
INST29					
INST30				O5	O3,O4
INST31					

INST32					
INST33					
INST34					
INST35					O5

The kernel is INST15 to INST19 that repeats over different iterations.

ASSIGNMENT 3

3A)
Assumptions: 2 FP add units with 2 cycle latency, 1 FP division unit with 5 cycle latency

O1: ADDD F1, F2, F3
O2: DIVD F4, F1, F2 → *wait for F1*
O3: SUBD F2, F4, F6 → *wait for F4*
O4: ADDD F4, F1, F1 → *wait for F1*

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13
O1	Issue	Exec	Exec	CDB *(F1)*									
O2		Issue	Issue	Issue	Exec	Exec	Exec	Exec	Exec	CDB *(F4)*			
O3			Issue	Issue	Issue	Issue	Issue	Issue	Issue	Issue	Exec	Exec	CDB *(F2)*
O4				Issue	Exec	Exec	CDB *(F4)*						

1. O1 flows through the pipeline without encountering any hazards
2. O2 has a RAW hazard with respect to O1 and cannot execute until cycle 5 when the result from O1 is broadcast over the CDB.
3. O3 has a RAW hazard with respect to O2 and cannot execute until cycle 11
4. O4 has a RAW hazard with respect to O1 and cannot execute before cycle 5
5. WAR (O3 with respect to O2) and WAW (O4 with respect to O2) hazards are resolved through renaming. Note that O2 will not write back to the register-file as register F4 is linked to the destination operand of O4. (See section 3.4.1 of the text book)

3B)

```
BNEZ    R1, LABEL
ADDD   F1, F2, F3              ;- O1
DIVD    F4, F1, F2             ;- O2
SUBD   F2, F5, F6              ;- O3
```

The reorder buffer (ROB) is the main mechanism to keep track of register values when instructions are speculatively executed. It buffers speculatively executed instructions **in the order they appear in the program,** that is, in program order. Each entry has room for the status of each instruction whether it is speculatively executed or committed. When an instruction is committed, it will be removed from the reorder buffer in the next cycle. When an instruction is speculatively executed, the entry also contains the value of the destination register, if it is available.

Now, when the branch instruction is validated as correctly predicted, it will be removed from the ROB in the next cycle. This happens, according to the assumptions, when the last instruction has been executed.

Let's make a pipeline diagram to track the execution of the last three instructions:

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
O1	Issue	Exec	Exec	CDB							
O2		Issue	Issue	Issue	Exec	Exec	Exec	Exec	Exec	CDB	
O3			Issue	Exec	Exec	CDB					

We note that from the point the branch instruction has retired from the ROB (in C7 according to the assumptions), it takes another **four cycles** until the result is written back to the register-file.

3C) See textbook, page 121 with respect to Figure 3.19.

ASSIGNMENT 4

4A)

- *Show how the code is annotated with prefetch instructions to hide all cache misses.*

According to the assumptions a loop iteration containing a prefetch instruction (CPI=2) and five other instructions (CPI=1) takes 7 cycles assuming that all load instruction hits. However, one of the five instructions is a load instruction and since the block size is four words, the load instruction in every fourth iteration misses. A cache miss takes 100 cycles. The question is how many iterations in advance we must launch a prefetch instruction.

i x 7 > 100 ⇒ i = 15

```
for (i=0; i<1000; i++){
     prefetch(A[i+15]
     C+=A[i];}
```

- *How many MSHRs are needed to make software prefetching maximally effective?*

We need 15 MSHRs because there are 15 outstanding prefetch instructions. However, since only every fourth load will miss we could optimize the program to launch a prefetch instruction only once every fourth iteration. This is however beyond the scope of the assignment.

- *How much faster does the program run on the system with a non-blocking cache using software prefetching?*

The program without prefetching:

Consider four iterations as there is a miss every fourth iteration:

Four iterations execute 4 x 4 + 3= 19 instructions that take a single cycle and 1 load instruction that takes 100 cycles. In total: 119 cycles

The program with prefetching:

Four iterations execute 4 x 5 = 20 instructions and four prefetch instructions that take two cycle each so 4 prefetch instructions that take 8 cycles. In total: 28 cycles

4B)

Let's calculate the miss rates. In one thousand instructions 10% are memory instructions, that is, 100. So in 100 instructions an infinite-sized cache experience 2 misses. This is a miss rate of 2%. So we have the following miss rates:

Cache organization	**MR**
16-KB direct mapped cache	10
16-KB 2-way assoc. cache	8
16-KB fully associative cache	6
Cache with infinite size	2

The cold miss rate is 2%, the capacity miss rate is (6-2=) 4% and the conflict miss rate is (10-4 -2=) 4%.

4C) See textbook, page 209

ASSIGNMENT 5

5A)

R_1
R_2
$W_{1=}0$
$W_{2=}1$
R_1
R_2

In a write-through cache, the memory is updated but not the content of any cache that has a copy of that block. The first two reads from P1 and P1 creates copies of the original content of the block. The next two writes, from P1 and P2 updates their respective private caches and memory with the values 0 and 1, from P1 and P2, respectively, in that order. Finally, the read from P1 returns the 0, not 1, which is incorrect as the last write, in the order, writes 1, not 0, breaking the correctness. The first write, by P1, should have invalidated the block in P2's cache.

5B) See textbook on page 254.

I->S: Any cache miss will result in a read request on the interconnect (bus in this case) that will return an up-to-date copy from memory.

S->I: This state transition is triggered by another processor that writes to a block that is present (Upgrade) or not present (BusRdX) in its cache. It will result in invalidating the local copy in the cache resulting in downgrading the block to Invalid (I).

5C) See textbook, pages 438-439 in the textbook.