Final Exam Solution - Computer Architecture and Operating Systems | ECE 3055

ECE3055B Fall 2004
Computer Architecture and Operating Systems
Final Exam Solution
Dec 10, 2004

1. (15%) General Q&A. Give concise and brief answer to each of the following questions.

1.1. (2%) What are the differences between a write-allocate and no-write-allocate policy in a cache?

Write-allocate: When a write (store) miss occurs, the missed cache line is brought into the first level cache before actual writing takes place.

No Write-allocate: When a write miss occurs, the memory update bypasses the cache and updates the next level memory hierarchy where there is a hit.

1.2. (2%) Explain what is "Control Hazard" and how to avoid it?

Control hazard: Instruction fetch depends on some in-flight instruction being executed. For example, the target address of a branch or jump is not immediately available after the branch/jump exits from fetch stage.

Solutions:
1. Design a hazard unit to detect the hazard and stall the pipeline.
2. branch prediction
3. using delay slot in the instruction scheduling

1.3. (2%) What is the primary advantage of adding a translation lookaside buffer (TLB)?

A specially tailored hardware designed for accelerating address translation from virtual address space to physical address space.

1.4. (4%) The following two functions represent one semaphore implementation for synchronization. However, it is known that the performance of such a solution suffers from CPU idling when no semaphore is available. Please re-implement these two semaphore functions that eliminate the spin busy waiting loop and can yield the idle waiting time to other processes which can perform useful work.

```
acquire(S) {                          release(S) {
          while (S<=0)                          S++;
                    ; // spin             }
loop
          S--;
}
```

Solution Code:

```
acquire(S) {                          release(S) {
          value--;                              value++;
          if (value <0) (                       if (value<=0) {
                    add process to wait list;             remove P from wait list
                    block;                                wakeup(P);
          }                                     }
}                                     }
```

1.5. (5%) The following routine (from textbook) is used to ensure mutual exclusion for 2 threads, t0 and t1. Two flags, flag0 and flag1 are initialized to 0 for t0 and t1, respectively. Describe a scenario in which the following mutex implementation leads to infinite waiting (or a deadlock) for both threads.

```
void enteringCriticalSection(int pid) {
          if (pid==0) {
                    flag0 = 1;
                    while (flag1) {
                              thread.yield();
                    }
          }
          else {
                    flag1 = 1;
                    while (flag0) {
                              thread.yield();
          }
}
```

The mutex fails when a context switch occurs after t0 sets flag0=1, and then t1 sets flag1=1. At this moment, both flags are set to 1. Thereafter, both threads t0 and t1 will yield indefinitely, i.e. a deadlock and no one will actually enter the critical section.

2. (10%) Amdahl's Law.

2.1. (8%) Assume we make an enhancement to a computer that improves some mode of execution by a factor of 15. This new fast mode is used 40% of the time, measured as a percentage of the execution time **when the fast mode is in use**. What is the overall speedup we can achieve?

$$\frac{0.4*15+(1-0.4)}{0.4+(1-0.4)} = 6.6$$

2.2. (2%) Continue from question 2.1, what percentage of the original execution time has been converted to fast mode?

Using Amdahl's Law

$$\frac{1}{\frac{x}{15}+(1-x)} = 6.6 \qquad \therefore\ x = 90.9\ \%$$

OR an easier way to compute it

$$\frac{0.4*15}{0.4*15+0.6}*100\% = 90.9\%$$

3. (25%) Instruction Set Architecture.

3.1. (10%) Given the following MIPS program being executed in a 5-stage pipeline as shown in next page. 0x10010000 is the address of the first instruction and each instruction is 4 bytes. The corresponding data and control signals are provided in the table next page. At the end of clock cycle 5 when all instructions are in the pipeline, fill the blanks with the resulting values in **hexadecimal**. Assume the registers are all loaded with the register number prior to execution and each data memory **byte** is loaded with the **least significant byte** of its own address. For example, 0x00000001 contains 0x01, 0x00000002 contains 0x02, 0x0000000F contains 0x0F, and so on. Thus, if lw from 0x00000000, one will get 0x03020100; lw from 0x00000004 will get 0x07060504. And there is no branch prediction hardware.

```
0x10010000:      lw    $9, ($8)
                 addi $11, $10, -2
                 xor  $16, $14, $15
                 sw   $17, ($21)
                 j    0x30030004
```

<u>Data signals</u>

Read data 1 in ID = 0x00000015________

Read data 2 in ID = 0x00000011__________

Write data in ID = 0x0b0a0908_______

ALU output in EX = 0x0000001d____

Input value to PC = 0x00000014_____________

MEMpass = 0x0000000b____________________

<u>Pipeline Control signals</u>

WB in ID/EX for EX = 0x2__________

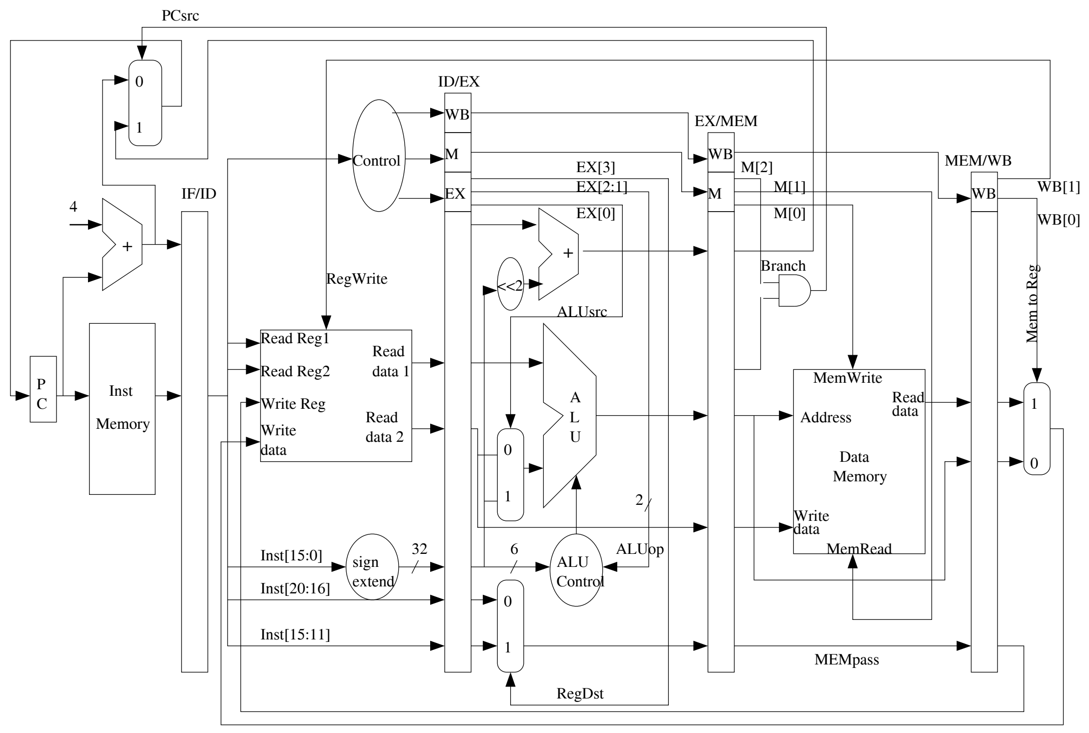
WB in MEM/WB for WB = 0x3__________

M in EX/MEM for MEM = 0x0______________

EX in ID/EX for EX = 0xC____________

Table 1. Control signal table

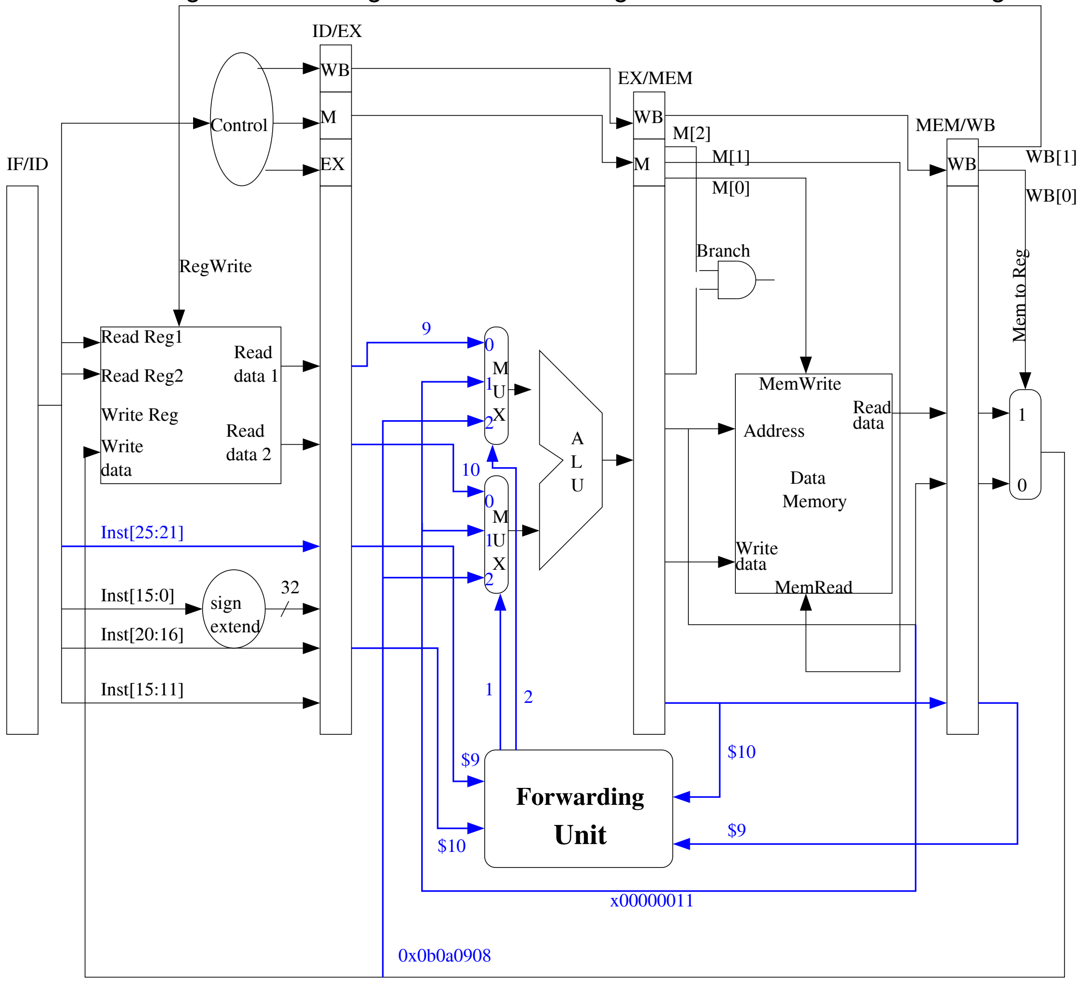
Instruction	**EX[3:0]**				**M[2:0]**			**WB[1:0]**	
	RegDst	**ALU op1**	**ALU op2**	**ALU src**	**Branch**	**MemRead**	**MemWrite**	**RegWrite**	**Mem toReg**
R-format	1	1	0	0	0	0	0	1	0
I-format	1	1	1	1	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	x	0	0	1	0	0	1	0	x
j	x	0	1	0	1	0	0	0	x



3.2. (10%) Now the code is changed to the sequence below. Assume there is no stall caused by memory/cache misses. Due to the data hazards, the processor needs a forwarding logic to handle true dependency without compromising performance. The pipeline is simplified to the relevant part for forwarding. Please **draw** the **necessary signals (1) going into** the forwarding logic and **(2) generated by** the logic. Also **draw** the **data signals** that **(3) go into** the input muxes of the ALU. Please **(4) label** all your signals with the correct data values (for only those used) based on the given code.

```
0x10010000:     lw   $9, ($8)
                addi $10, $19, -2
                xor  $11, $9, $10
                sw   $10, ($8)
                j    0x3003000
```

Here I only show the necessary signals for the code provided
The forwarding unit takes those register ID and compare them to generate the correct MUX control signals for routing either data from register file or from the later 2 stages.



3.3. (5%) Now we store the first 4 instructions in memory in the original order, please fill the memory contents (in **hex**) below for (1) a Little Endian system; (2) a Big Endian system. (Note that each row is a single byte)

```
                                                     Encoding
0x10010000:      lw    $9, ($8)               0x8d090000
                 addi $10, $9, -2             0x212afffe
                 xor  $11, $9, $10            0x012a5826
                 sw   $10, ($8)               0xad0a0000
```

Little Endian	Low memory address	Big Endian
00	0x10010000	8d
00	0x10010001	09
09	0x10010002	00
8d	0x10010003	00
fe	0x10010004	21
ff		2a
2a		ff
21		fe
26		01
58		2a
2a		58
01		26
00		ad
00		0a
0a		00
ad	0x1001000F	00
	High memory address	

4. (20%) Given the following address access stream, please fill the blanks. All the addresses are 32-bit. Please note that if there is any replacement, you can either cross out or erase the original content and fill in the new one. Bottom line is that you have show the final tag content inside the cache. Use **Hex value** for the Tag.

```
read  0xBBCCD288
read  0xBBCCDF88
write 0xBBCCD300
write 0xBBCCDFFC
read  0xBBCCD680
```

4.1. (10%) A 1 kilo-byte, direct-mapped writeback cache. The cache line size is 128 bytes. Please first draw and finish the cache with the valid bit, dirty bit and tag array with the ***right*** number of sets. (One example cache line is shown below.) Then start to fill the correct values for V, D and Tag based on the access stream above.

V	D	TAG
1	0	2EF335
1	1	2EF334
1	1	2EF337

Final snapshot

1KB/128 = 8 sets

<u>sequence</u>
After 1st read: 2EF334 in set 5, V=1

After 2nd read: 2EF337 in set 7, V=1

After 3rd write: 2EF334 in set 6, V=D=1

After 4th write: set 7 D =1 (a cache hit)

After 5th read: 2EF335 replaces 2EF334 in set 5, V=1

4.2. (10%) A 2 kilo-byte, 2-way writeback cache. The cache line size is 256 bytes. Please first draw and finish the cache with the valid bit, dirty bit and tag array with the ***right*** number of sets. Please also **number the way** in your drawing. (One example cache line is shown below.) Then start to fill the correct values for V, D and Tag based on the access stream above.

2KB/(256*2) = 4 sets for each way

Way 0

V	D	TAG
1	0	2EF334
1	1	2EF337

Way 1

V	D	TAG
1	0	2EF335
1	1	2EF334

After 1st read: 2EF334 in set 2, way 0, V=1

After 2nd read: 2EF337 in set 3, way 0, V=1

After 3rd write: 2EF334 in set 3, way 1, V=D=1

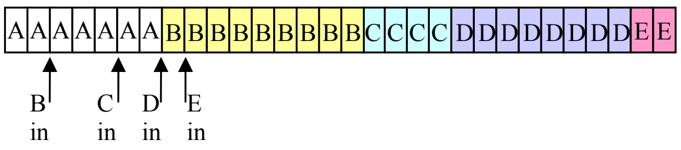
After 4th write: set 3 of way 0, D= 1 (a cache hit)

After 5th read: 2EF335 in set 2 of way 1, V=1

5. (20%) Answer the following questions based on the table below.

Process ID	Arrival Time	Burst Time
A	0	7
B	2	9
C	5	4
D	7	8
E	8	2

5.1. (5%) Show your schedule with timeline and Calculate the average **waiting** time when use **First-Come First-Serve (FCFS)** scheduling. (Please take arrival time into account.)



Waiting Time

A = 0

B = 5

C = 11

D = 13

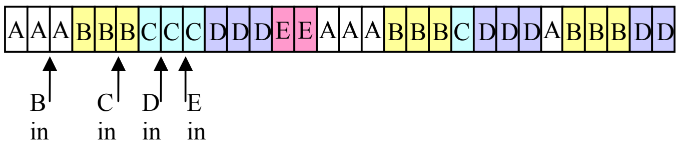
E = 20

Average = 49/5 = 9.8

5.2. (5%) Show your schedule with timeline and Calculate the average **turnaround** time when use **Round-Robin (RR)** scheduling with time quantum 3. (Please take arrival time into account.)

Process ID	Arrival Time	Burst Time
A	0	7
B	2	9
C	5	4
D	7	8
E	8	2

This is the same table, shown here for your convenience.



Turnaround Time

A = 25

B = 26

C = 16

D = 23

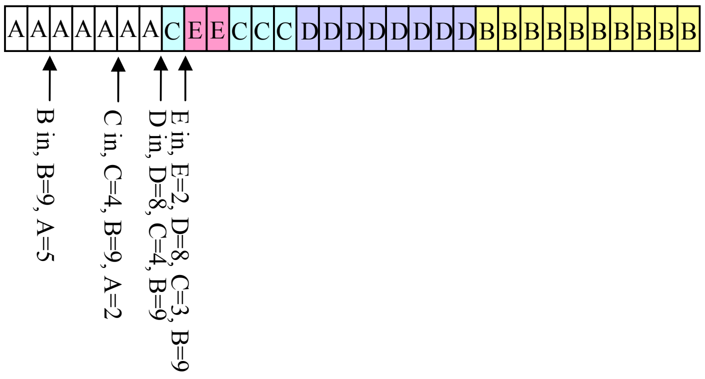
E = 6

Average = 96/5 = 19.2

5.3. (5%) Show your schedule with timeline and Calculate the average **waiting** time when use **Shortest-Remaining-Time-First scheduling with preemption.** (Please take arrival time into account.)

Process ID	Arrival Time	Burst Time
A	0	7
B	2	9
C	5	4
D	7	8
E	8	2

This is the same table, shown here for your convenience



Waiting Time

A = 0

B = 19

C = 4

D = 6

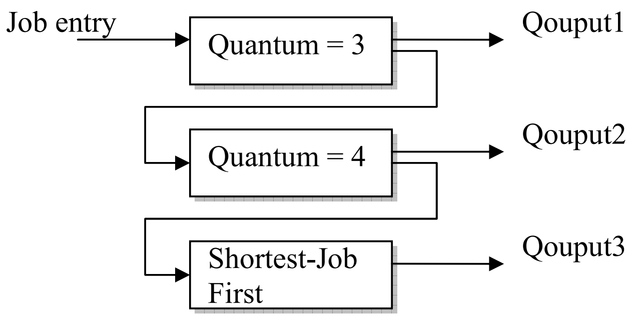
E = 0

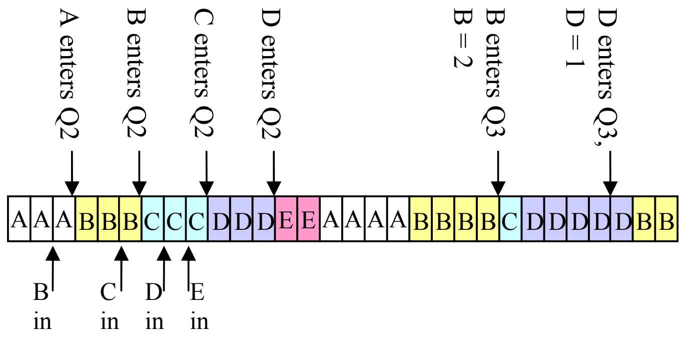
Average = 29/5 = 5.8

5.4. (10%) Show your schedule with timeline and Calculate the average **turnaround** time when use the **multi-level feedback queue** as below. (Please take arrival time into account.) Note that the priority of the top 2 queues is based on arrival times.

Process ID	Arrival Time	Burst Time
A	0	7
B	2	9
C	5	4
D	7	8
E	8	2

This is the same table, shown here for your convenience





Turnaround Time

A = 18

B = 28

C = 18

D = 21

E = 6

Average = 91/5 = 18.

6. (10%) Deadlock Avoidance.
Given the following snapshot of a system and five current processes. The Allocation, Max, and Available represent the number of resources allocated to each process, the maximum number of resources needed by each process, and the available number of resources based on the current allocation. Pleas examine if the request (0, 1, 1, 0) for (A, B, C, D) made by P2 can be granted for not entering a deadlock situation? If it can be granted, please find and show the safe sequence. You have to show all your work to get credit.

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	5	3	1	0	8	5	2	0	2	2	1
P1	1	1	1	0	1	6	4	2				
P2	0	1	2	0	0	3	5	0				
P3	3	1	2	1	3	1	2	2				
P4	0	1	2	3	1	5	4	6				

If we allocate (0,1,1,0) to P2, then Allocation of P2 will become (0,2,3,0) and Available will drop to (0,1,1,1). Based on which, we generate the new matrix below with the maximum Need of each process for finding the safe sequence using Banker's algorithm where Need = Max - Allocation

	Need				Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	3	2	1	0	5	3	1	0	8	5	2	0	1	1	1
P1	0	5	3	2	1	1	1	0	1	6	4	2				
P2	0	1	2	0	0	2	3	0	0	3	5	0				
P3	0	0	0	1	3	1	2	1	3	1	2	2				
P4	1	4	2	3	0	1	2	3	1	5	4	6				

With (0, 1, 1, 1) left, only P3 can be satisfied if requested up to the max by any Pi. After P3 is done, Available = (0,1,1,1)+(3,1,2,1) = (3,2,3,2)

With (3,2,3,2), the next process can be satisfied is P2 only. After P2 is done, Available = (3,2,3,2)+(0,2,3,0) = (3,4,6,2)

With (3,4,6,2), the next process can be satisfied is P0 only. After P0 is done, Availability = (3,4,6,2)+(0,5,3,1) = (3,9,9,3)

With (3,9,9,3), the next process can be satisfied can be either P1 or P4.

Thus we found 2 possible safe sequences:
<P3, P2, P0, P1, P4> or < P3, P2, P0, P4, P1>