Department of Electrical and Computer Engineering

The University of Texas at Austin

EE 460N, Fall 2016
Problem Set 3 Solution
Yale N. Patt, Instructor
Siavash Zangeneh, Ali Fakhrzadehgan, Steven Flolid, Matthew Normyle, TAs

1. An 8KB cache size with a 8B line size, in a 4-way set associative cache means there are **8KB ÷ (4 × 8B) = 256** sets in the cache.

   Since there are 256 or 2^8 sets, 8 bits are required to index into the correct set. Since there are 8B or 2^3 bytes in a cache line, 3 bits are required to find the correct byte within a block. Given a 24-bit address space, this leaves **24 - 8 - 3 = 13** bits left over for the tag store. Additionally, the tag store must hold 2 bits for the V/NV replacement policy and 1 valid bit. This means each cache line must have a 16-bit tag store associated with it. 2B of tag store times (256 × 4) cache lines in the cache means that the tag store, in total takes up 2048 bytes, which is 16384 bits

2. The size of the tag store is $\mathbf{2^{12} + 2^8}$. We know that the size of the tag store can be given as the product of number of sets and number of bits per set.

   We also know that address space is 16-bits. Hence, ***tag* + *index* + *bib* = 16**

   In order to find *bib*, we need to find *index* and *tag*. The following bits are necessary for each set of the tag store:

   - 1 bit for LRU (remember: you only need one bit per set for a 2-way associative cache)
   - 2 valid bits
   - 2 dirty bits
   - 2 × *tag* tag bits

   Total number of bits per set is **5 + 2 × *tag***. An important conclusion that can be drawn is that number of bits per set will always be an odd number.

   We will also use the fact that number of sets is always a power of 2 (since it is indexed using an integer number of bits). Given the size of the tag store, *index* has to be a number less than 8 (the size of the tag store is indivisible by 2^9 or greater).

   The only value of *index* that fits both criterion is 8. Therefore:

   Number of sets = 2^8 = 256

   Bits per row = 4352 ÷ 256 = 17

   17 = 5 + 2 × *tag* ⇒ *tag* = 6

   6 + 8 + *bib* = 16 ⇒ *bib* = 2

   Hence, the cache block size is 4 bytes

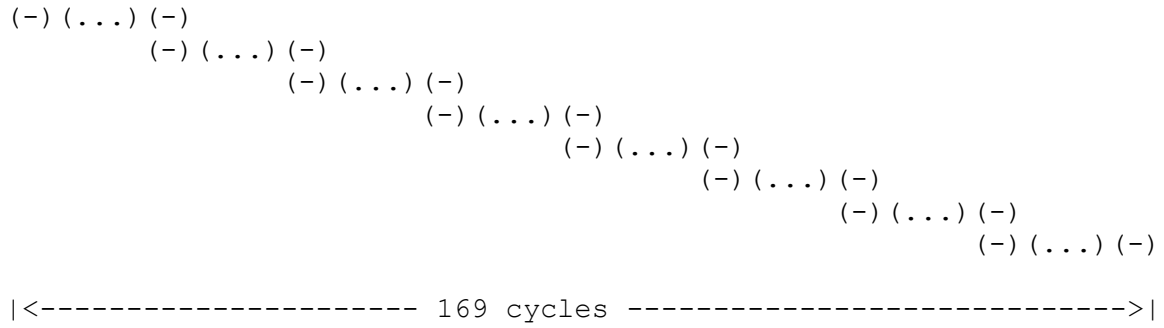
3. ***average_access_time_per_instruction* = 1 × instruction_access_time + 0.3 × data_access_time**

   We can use the following equations in both part a and part b to compute the instruction and data access time.

   ***instruction_access_time* = 1 + 0.05 × (6 + 0.15 × mem_latency)**

   ***data_access_time* = 1 + 0.10 × (6 + 0.25 × mem_latency)**

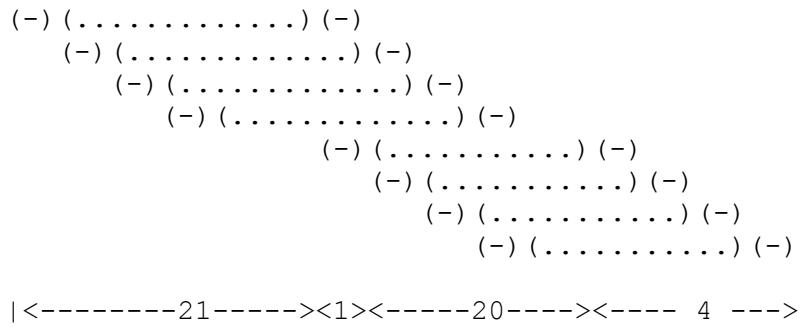
   1. We need to calculate the memory latency. Since each cache block is 8 words, it will take 8 accesses to get a cache block from memory.



Using the equations, we get 2.5675 cycles for *instruction_access_time* and 5.825 for *data_access_time*.

The average latency is **2.5675 + 0.3 × 5.825 = 4.315 cycles**

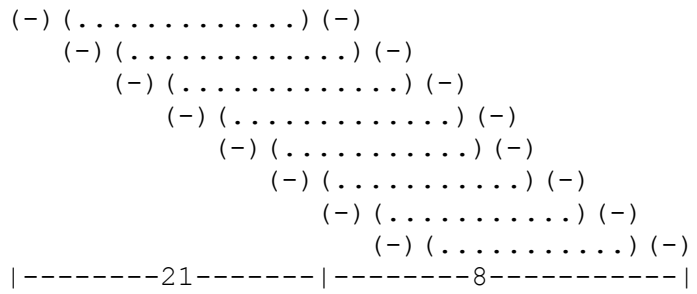
2. We again need to compute the memory latency:



So the mem_latency is 46 cycles. Again using the equations, the instruction_access_time is 1.645 and data_acces_time is 2.75.

The average latency is **1.645 + 2.75 × 0.3 = 2.47 cycles**

3. We need to compute memory latency for 8-way interleaving



So memory latency is 29 cycles. Using the equations, we get the instruction_access_time as 1.517 and data_acces_time as 2.325.

The average latency is 1.517 + .3 * 2.325 = 2.215

4. For the 4-way interleaved memory, the improvement is **(4.315 - 2.47) ÷ 4.315 = 0.4276** Therefore, the average latency improves by 42.76%

   For the 8-way interleaved memory, the improvement is **(4.315 - 2.215) ÷ 4.315 = 0.4867** Therefore, the average latency improves by 48.67%

4. We assume that the address is 12 bits (we could also have assumed a 10-bit address)

   a. The address is divided into 3 portions:

   - address[1:0] (2 bits) for the byte in the block

- address [4:2] (3 bits) for the cache index
- address[11:5] (7 bits) for the tag

The contents of the cache at the end of each pass are the same. They are shown below:

Valid	Tag	Data (addresses are written inside each byte)			
1	0010 000	203	202	201	200
1	0010 000	207	206	205	204
1	0010 000	20B	20A	209	208
1	0010 010	24F	24E	24D	24C
1	0010 111	2F3	2F2	2F1	2F0
1	0010 111	2F7	2F6	2F5	2F4
1	0010 000	21B	21A	219	218
1	0010 000	21F	21E	21D	21C

The hit/miss information for each pass is shown below:

Reference	200	204	208	20C	2F4	2F0	200	204	218	21C	24C	2F4
Pass 1:	M	M	M	M	M	M	H	H	M	M	M	H
Pass 2:	H	H	H	M	H	H	H	H	H	H	M	H
Pass 3:	H	H	H	M	H	H	H	H	H	H	M	H
Pass 4:	H	H	H	M	H	H	H	H	H	H	M	H

Hit rate is 33/48

b. The address is divided into two: 2 bits for identifying the byte in the block, 10 bits for the tag. No bits are needed for cache index. The following table shows the contents of the cache at the end of each pass (Valid bits and Tags are ignored, they should be obvious. Starting addresses of blocks in the cache are provided):

Pass	Way 0 Data	Way 1 Data	Way 2 Data	Way 3 Data	Way 4 Data	Way 5 Data	Way 6 Data	Way 7 Data
1	200	204	24C	20C	2F4	2F0	218	21C
2	200	204	21C	24C	2F4	20C	2F0	218
3	200	204	218	21C	2F4	24C	20C	2F0
4	200	204	2F0	218	2F4	21C	24C	20C

The hit/miss information for each pass is shown below:

Reference:	200	204	208	20C	2F4	2F0	200	204	218	21C	24C	2F4
Pass 1:	M	M	M	M	M	M	H	H	M	M	M	H
Pass 2:	H	H	M	M	H	M	H	H	M	M	M	H
Pass 3:	H	H	M	M	H	M	H	H	M	M	M	H
Pass 4:	H	H	M	M	H	M	H	H	M	M	M	H

Hit rate is 21/48

c. The address is divided into 3 portions: 2 bits for identifying the byte in the block, 1 bit for the cache index, 9 bits for the tag. The contents of the cache at the end of Pass 1:

Way 0			Way 1			Way 2			Way 3		
V	**Tag**	**Data**	**V**	**Tag**	**Data**	**V**	**Tag**	**Data**	**V**	**Tag**	**Data**

1	0010 0000 0	203-200	1	0010 0000 1	20B-208	1	0010 1111 0	2F3-2F0	1	0010 0001 1	21B-218
1	0010 0000 0	207-204	1	0010 0100 1	24F-24C	1	0010 1111 0	2F7-2F4	1	0010 0001 1	21F-21C

The hit/miss information for each pass is shown below:

Reference:	200	204	208	20C	2F4	2F0	200	204	218	21C	24C	2F4
Pass 1:	M	M	M	M	M	M	H	H	M	M	M	H
Pass 2:	H	H	H	M	H	H	H	H	H	M	M	H
Pass 3:	H	H	H	M	H	H	H	H	H	M	M	H
Pass 4:	H	H	H	M	H	H	H	H	H	M	M	H

Contents of the second set of the cache (index equals 1) after pass 2, 3, and 4 (the first set remains the same):

Pass	**Way 0**			**Way 1**			**Way 2**			**Way 3**		
	V	**Tag**	**Data**	**V**	**Tag**	**Data**	**V**	**Tag**	**Data**	**V**	**Tag**	**Data**
2	1	0010 0000 0	207-204	1	0010 0001 1	21F-21C	1	0010 1111 0	2F7-2F4	1	0010 0100 1	24F-24C
3	1	0010 0000 0	207-204	1	0010 0100 1	24F-24C	1	0010 1111 0	2F7-2F4	1	0010 0001 1	21F-21C
4	1	0010 0000 0	207-204	1	0010 0001 1	21F-21C	1	0010 1111 0	2F7-2F4	1	0010 0100 1	24F-24C

Hit Rate is 30/48

5. ## Block Size

The following table lists hit ratios for different cache block sizes given the access sequence 1 (0, 2, 4, 8, 16, 32):

Block Size	**Hit Ratio**
1B	0/6
2B	0/6
4B	1/6
8B	2/6
16B	3/6
32B	4/6

Since the hit ratio is reported as 0.33 for this sequence, the block size must be 8 bytes. Therefore, the accesses look like this:

Address	0	2	4	8	16	32
Hit/Miss	M	H	H	M	M	M

Associativity

Notice that all of the addresses of sequence 2 (0, 512, 1024, 1536, 2048, 1536, 1024, 512, 0) are multiples of 512. This means that they all map to the same set, since the size of the cache can only be 256 or 512 bytes. Therefore, the hit ratio would be 0/9 in case of a direct-mapped cache. In case of a 2-way cache, the accesses would behave as follows:

Address	0	512	1024	1536	2048	1536	1024	512	0
Hit/Miss	M	M	M	M	M	H	M	M	M

The resulting hit rate would be 1/9. Similarly, for 4-way cache:

Address	0	512	1024	1536	2048	1536	1024	512	0
Hit/Miss	M	M	M	M	M	H	H	H	M

The resulting hit rate would be 3/9, and thus the cache is 4-way set associative.

Cache Size

If the cache size were 256B, there would be 3 index bits and all of the adresses in sequence 3 (0, 64, 128, 256, 512, 256, 128, 64, 0) would map to the same set:

Address	0	64	128	256	512	256	128	64	0
Hit/Miss	M	M	M	M	M	H	H	H	M

The resulting hit ratio would be 3/9, and thus the cache size is 256B. For completeness, here's how the accesses would look like in case of a 512B cache (4 index bits):

Address	0	64	128	256	512	256	128	64	0
Hit/Miss	M	M	M	M	M	H	H	H	H

Replacement Policy

In case of the FIFO replacement policy, the accesses of sequence 3 (0, 512, 1024, 0, 1536, 0, 2048, 512) would look as follows:

Address	0	512	1024	0	1536	0	2048	512
Hit/Miss	M	M	M	H	M	H	M	H

The resulting hit ration would be 3/8. However, in case of the LRU replacement policy the hit rate would be 0.25:

0	512	1024	0	1536	0	2048	512
M	M	M	H	M	H	M	M

Thus the replacement policy is LRU.

6. Differences between exceptions and interrupts:

	Exceptions	**Interrupts**
Cause	Internal to the running process	External to the running process
When are they handled?	When detected (mostly)	When convenient (mostly)
Are they maskable?	Almost never	Almost always
Priority	Same as the process that caused the exception.	Depends (on the priority of the interrupting device)
Context	Process	Handling is done withing the system context.

Interrupts and exceptions are similar in how they are handled. To handle both, the machine has to be put into a consistent state and PC needs to be loaded with the address of the interrupt/exception handler.

Basic steps required to handle an exception/interrupt:

1. Exception/interrupt is detected.

2. The machine is put to a consistent state.
3. PC and PSR (Program Status Register) of the process are saved (on the stack).
4. PC is loaded with the address of the handler. (This address can be obtained by accessing the interrupt/exception vector table using the vector supplied by the event)
5. Interrupt/exception handler is executed.
6. Interrupt/exception handler returns back to the interrupted process. (PC and PSR of the interrupted process are restored)

7. The state diagram for the FSM device controller can be found in the handouts section of the website under the notes for I/O. Note that in the notes, the transition from BGout to IDLE is based on the SACK signal. This is to illustrate the second race condition which is corrected by basing the transition on (NOT BGin).

   The input and output signals of the controller are:

Signal	**Type**	**Function**
D	Input	Asserted when the device needs to initiate a bus transaction
BG_{in}	Input	Incoming bus grant signal, asserted by the priority arbitration unit
$BBSY_{in}$	Input	Asserted by the current bus master. Negative edge indicates the end of a bus cycle.
MSYN	Input	Master-side handshaking signal that controls the bus transaction between the bus master and the slave
SSYN	Input	Slave-side handshaking signal that controls the bus transaction between the bus master and the slave
BR_{out}	Output	Asserted to request the bus. Goes to the priority arbitration unit.
SACK	Output	Asserted by the device that has won the arbitration
$BBSY_{out}$	Output	Same as $BBSY_{in}$
BG_{out}	Output	Asserted when the controller needs to pass the bus grant signal down the daisy chain.

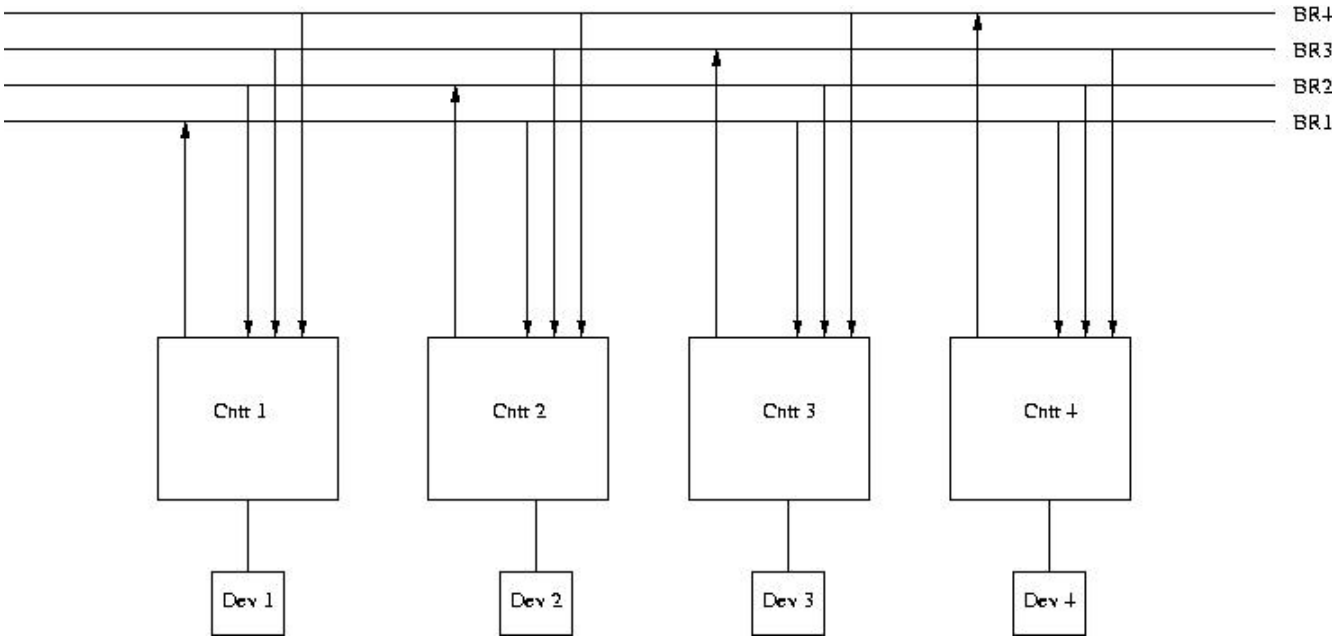
   Two race conditions are:

   1. This race condition is subtle. From the PAU side the PAU asserts BGj, which works its way down the daisy chain to all devices at BRj. Before SACK is asserted, a controller asserts BRk, where k is higher priority than j. PAU now asserts BGk, and you have two BG signals propagating, which will result in two controllers thinking they are the next bus master.

      Solution: PAU latches BR signals when it sees NOT-BBSY, indicating it is okay to grant the bus again. NOT-SACK is also gated (after sufficient delay) with the BG signals, guaranteeing that a BG signal cannot be asserted until after PAU logic has taken effect. Any subsequent BR signal does not get latched and so can not affect the PAU logic. We also discussed that using NOT-BBSY as load-enable can be problematic if the bus is not used for a long preiod. By using NOR(BG0, BG1,...) as load-enable, we solved this issue.

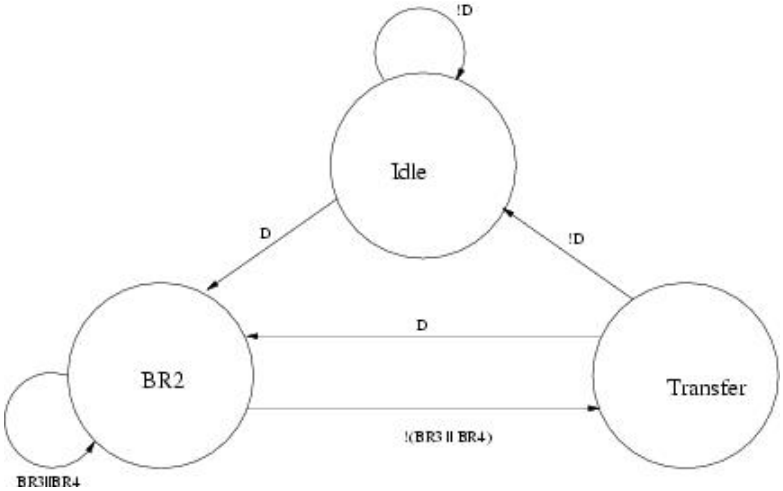
   2. Let's say device controller D1 is in BG_{out} state. This means that some device D2 that is down the same daisy chain as D1 had requested and is granted the bus. Let's say the device of D1 asserts the D signal while D1 is in BG_{out} state. D2 will eventually receive the BG_{in} signal and transition to the SACK state. It will take some time for the SACK signal to travel to the priority arbitration unit. The SACK signal probably reaches D1 before it reaches the priority arbitration unit. Hence, when the SACK signal reaches D1 the BG_{in} input of D1 is still being asserted. Therefore, upon receiving the SACK signal D1 will immediately transition to IDLE to BR_{out} to SACK states. Hence, both D1 and D2 will be asserting the SACK signal which is not desirable. A simple solution that fixes this race condition is not transitioning to IDLE state if the BG_{in} signal is still high.

8. a. The solution is shown below:



b. It is possible. Consider a case where device 3 and 4 just alternate the bus.

c. The solution is shown below.



9. Moved to PS4
10. Moved to PS4
11. Moved to PS4
12. Moved to PS4
13. Moved to PS4
14. Moved to PS4
15. Moved to PS4

16. Single Error. The corrected bit pattern:

```
111010010111
```

When there are two errors, there will definitely be a parity error. However, consider the case where P0 and P1 are the two errors. If we examine the parity errors with the intention of attempting to correct a single error as in the previous example, then we would erroneously think that the 3rd bit (D0) was in error. Clearly, this is not the case. It is also possible that after computing the parity errors, we determine, for example, that bit 15 is in error (all four parity functions evaluated incorrectly). However, the transmitted data only has 12 bits. Thus, if we see that an error points to bits 13 through 15, we know for sure that two errors occurred. However, as described earlier, it is definitely possible that two errors manifest themselves as a single error in one of the 12 bits transmitted.

More than two errors will manifest itself as a single error, two errors, or possibly even no error at all. This scheme has no way to distinguish more than 2 errors from 2 errors or less.