# Department of Electrical and Computer Engineering

## The University of Texas at Austin

EE 460N, Fall 2016
Problem Set 2 Solutions
Yale N. Patt, Instructor
Siavash Zangeneh, Ali Fakhrzadehgan, Steven Flolid, Matthew Normyle, TAs

1. This problem has been postponed to later problem sets

2. This problem has been postponed to later problem sets

3. Truth table for the WE Logic:

| MAR[0] | R.W | DATA.SIZE | WE1 | WE0 |
|---|---|---|---|---|
| 0 | RD | Byte | 0 | 0 |
| 0 | RD | Word | 0 | 0 |
| 0 | WR | Byte | 0 | 1 |
| 0 | WR | Word | 1 | 1 |
| 1 | RD | Byte | 0 | 0 |
| 1 | RD | Word | 0 | 0 |
| 1 | WR | Byte | 1 | 0 |
| 1 | WR | Word | 0 | 0 |

```
RD = 0  WR = 1;  Byte = 0  Word = 1
WE0 = (MAR[0]') AND (R.W)
WE1 = R.W AND (MAR[0] XOR DATA.SIZE)
```

4. a.
   - Byte on bus `Addr[1:0]`
   - Interleave bits `Addr[4:2]`
   - Chip address `Addr[7:5]`
   - Rank bits `Addr[11:8]`

   b. 577 Cycles. The first 8 memory accesses, A[0][0] to A[0][7], must occur sequentially with no overlap since they are all acesses to the same bank. Thus, it would take 80 cycles for the 1st 8 memory accesses, with the 8th access starting in cycle 70. Since the 8th and 9th memory accesses, A[0][7] and A[1][0], respectively, are to different banks, the accesses can overlap, and the 9th access can start in cycle 71 (70 cycles for the 1st 7 accesses plus 1 additional cycle of the 8th access). Continuing with this logic, the access to A[2][0] could start in cycle 142 (71x2). Finally, the access to A[7][0] could start in cycle 497 (71x7). Now all that remains are 8 more memory accesses, all to the same bank (A[7][0] to A[7][7]). This takes another 80 cycles, bringing the total to 577 cycles (497 + 80).

   If the memory were not interleaved, all 64 memory accesses must happen sequentially with no overlap, so it would take a total of 640 cycles (64*10). Therefore, we do gain some benefit from this interleaving scheme, but not that much.

   c. Yes, a change can be made. The new bits are:

   - Byte on bus `Addr[1:0]`
   - Interleave bits `Addr[4:2]`

- Chip address `Addr[11:9]`
- Rank bits `Addr[8:5]`

73 Cycles. With the new interleaving scheme, consecutive memory accesses are to either to different banks of the same rank, or to different ranks all together. In both cases, the consecutive accesses can start immediately after each other. Therefore the latency of all memory accesses would be hidden except the first access. Total number of cycles = 10 + 63 = 73

d. Only one line of code needs to be changed:

```
sum = sum + A[i][j];
to
sum = sum + A[j][i];
```

Alternatively, you could keep that line the same, but swap the variable (`i`/`j`) of the inner and outer loops as shown below.

Original code:

```
for(i = 0; i < 8; ++i){
  for(j = 0; j < 8; ++j){
    sum = sum + A[i][j];
  }
}
```

New code:

```
for(j = 0; j < 8; ++j){
  for(i = 0; i < 8; ++i){
    sum = sum + A[i][j];
  }
}
```

87 Cycles. Now consecutive memory accesses are to different banks, so the accesses can overlap. The 1st access, A[0][0], would begin at cycle 0, the 2nd, A[1][0], at cycle 1, and so on. The 8th access, A[7][0], would start at cycle 7. However, the 9th access, A[0][1], cannot start at cycle 8. It would have to wait 2 more cycles for the 1st access to finish since it is on the same bank as the 1st access; therefore, it would start at cycle 10. Continuing this logic, the access to A[0][2] would start at cycle 20, and finally, the access to A[0][7] would start at cycle 70. Now, all that is left are 8 accesses, but they are all to different banks so they can start 1 cycle after each other. The access to A[1][7] would begin at cycle 71, A[2][7] at 72, and finally A[7][7], the last memory access, would begin at cycle 77 and, therefore, end at cycle 87

5. In this problem, we assume that both of the rotators are right rotators. If the rotator for read is a right rotator and the rotator for write is a left rotator, PA[1:0] can be used as the control for both rotators.

Note: for WR (i.e. stores), the LD.MDR signal doesn't matter if the store takes only a single access, since we assume the MDR is already loaded with the data to be stored from the processor in the previous cycle. The only time LD.MDR matters is during the 1st access of a multi-access store; in this case, we need to make sure that the bytes that will be stored in the 2nd access aren't overwritten.

| PA[1:0] | SIZE | RD/WR | 1st/2nd | LD.MDR[3:0] | ROT[1:0] | WE[3:0] | sel |
|---|---|---|---|---|---|---|---|
| 00 | B | RD | X | XXX1 | 00 | 0000 | 0 |
| 00 | B | WR | X | XXXX | 00 | 0001 | 0 |
| 00 | H | RD | X | XX11 | 00 | 0000 | 0 |
| 00 | H | WR | X | XXXX | 00 | 0011 | 0 |

| PA[1:0] | SIZE | RD/WR | 1st/2nd | LD.MDR[3:0] | ROT[1:0] | WE[3:0] | sel |
|---|---|---|---|---|---|---|---|
| 00 | W | RD | X | 1111 | 00 | 0000 | 0 |
| 00 | W | WR | X | XXXX | 00 | 1111 | 0 |
| 01 | B | RD | X | XXX1 | 01 | 0000 | 0 |
| 01 | B | WR | X | XXXX | 11 | 0010 | 0 |
| 01 | H | RD | X | XX11 | 01 | 0000 | 0 |
| 01 | H | WR | X | XXXX | 11 | 0110 | 0 |
| 01 | W | RD | 1st | X111 | 01 | 0000 | 0 |
| 01 | W | RD | 2nd | 1000 | 01 | 0000 | 1 |
| 01 | W | WR | 1st | 0000 | 11 | 1110 | 0 |
| 01 | W | WR | 2nd | XXXX | 11 | 0001 | 1 |
| 10 | B | RD | X | XXX1 | 10 | 0000 | 0 |
| 10 | B | WR | X | XXXX | 10 | 0100 | 0 |
| 10 | H | RD | X | XX11 | 10 | 0000 | 0 |
| 10 | H | WR | X | XXXX | 10 | 1100 | 0 |
| 10 | W | RD | 1st | XX11 | 10 | 0000 | 0 |
| 10 | W | RD | 2nd | 1100 | 10 | 0000 | 1 |
| 10 | W | WR | 1st | 0000 | 10 | 1100 | 0 |
| 10 | W | WR | 2nd | XXXX | 10 | 0011 | 1 |
| 11 | B | RD | X | XXX1 | 11 | 0000 | 0 |
| 11 | B | WR | X | XXXX | 01 | 1000 | 0 |
| 11 | H | RD | 1st | XXX1 | 11 | 0000 | 0 |
| 11 | H | RD | 2nd | XX10 | 11 | 0000 | 1 |
| 11 | H | WR | 1st | XX00 | 01 | 1000 | 0 |
| 11 | H | WR | 2nd | XXXX | 01 | 0001 | 1 |
| 11 | W | RD | 1st | XXX1 | 11 | 0000 | 0 |
| 11 | W | RD | 2nd | 1110 | 11 | 0000 | 1 |
| 11 | W | WR | 1st | 0000 | 01 | 1000 | 0 |
| 11 | W | WR | 2nd | XXXX | 01 | 0111 | 1 |

Legend

| | |
|---|---|
| B(yte) | 00 |
| H(alf word) | 01 |
| W(ord) | 10 |
| RD(read) | 0 |
| WR(write) | 1 |
| 1st | 0 |
| 2nd | 1 |

6. Interleave into 64 banks in order to hide the latency in sequential accesses (note, minimum needed is 37 banks but one would really prefer to use a power of 2, therefore 64).

7. a. The probability of a single bit flipping is $p_f = 10^{-7}$. Therefore, the probability of a bit remaining correct is $p_c = (1 - 10^{-7})$. The probability that a transmitted nine bit message will have zero flipped bits is $p_c^9 = (1 - 10^{-7})^9$. Thus, the probability of at least one of the bits being flipped is $1 - p_c^9 = 1 - (1 - 10^{-7})^9 \approx 9 \times 10^{-7}$.

b. Parity check logic *can* detect an odd number of errors: 1, 3, 5, 7, and 9.

c. Parity check logic *cannot* detect an even number of errors: 2, 4, 6, 8.

d.
- There are choose(9, 1) = 9 possible combinations that result in a single flipped bit. Thus, the probability of one bit being flipped is $p_1 = 9 \times p_f \times {p_c}^8 \approx 9 \times 10^{-7}$.
- There are choose(9, 2) = 36 possible combinations that result in two flipped bits. Thus, the probability of two bits being flipped is $p_2 = 36 \times {p_f}^2 \times {p_c}^7 \approx 3.6 \times 10^{-13}$.
- There are choose(9, 3) = 84 possible combinations that result in three flipped bits. Thus, the probability of three bit being flipped is $p_3 = 84 \times {p_f}^3 \times {p_c}^6 \approx 8.4 \times 10^{-20}$.

e. Ignoring the probability of three or more bit errors, the probability of a detected error is just the probability of a single bit error (calculated above). Thus, the rate of detected errors is $p_1 \times 10^9 \div 9 \approx$ 100 errors per second.

f. Similarly, the probability of an undetected error is approximately the probability of a double error. Thus, the rate of undetected errors is $p_2 \times 10^9 \div 9 \approx 4 \times 10^{-5}$ corrupt messages per second (or twice as many undetected *bit* errors, since we assume each undetected corrupt message contains two flipped bits), which is equivalent to one undetected corrupt message about every 7 hours.

8.

| Reference | TLB hit | Page Fault |
|---|---|---|
| 0 | X | |
| 13 | X | |
| 5 | | |
| 2 | | |
| 14 | | X |
| 14 | X | |
| 13 | | |
| 6 | | X |
| 6 | X | |
| 13 | X | |
| 15 | | X |
| 14 | | |
| 15 | X | |
| 13 | X | |
| 4 | | X |
| 3 | | X |

TLB hit rate = 7/16.

TLB contains entries for pages 3, 4, and 13.

Solutions for the final contents of the frames of physical memory may differ slightly depending on what order the initially empty frames were allocated; however, no page should appear in more than one frame. Possible answers are shown below.

Frame 0 Page 14 (or 6 or 15)
Frame 1 Page 13
Frame 2 Page 3
Frame 3 Page 2
Frame 4 Page 6 (or 14 or 15)

Frame 5 Page 4
Frame 6 Page 15 (or 6 or 14)
Frame 7 Page Table

9. Size of a page is 512 bytes.

   Number of bits of address required to calculate the offset within a page is 9.

   Number of frames in physical memory is (8K bytes) ÷ (512 bytes) = $2^{13} \div 2^9 = 2^4$.

   Size of PFN is 4 bits.

   Size of PTE equals 1 (Valid) + 1 (Modified) + 2 (access control) + 4 (PFN) = 8 bits = 1 byte.

   Number of virtual pages in System Space is $(3 \times 2^{12}) \div (2^9) = 24$ pages.

   Size of System Page Table is 24 × 1 byte = 24 bytes = 24 × 8 bits = 192 bits.

10. We can determine from the given information that:

    - A Virtual Address (VA) is 16 bits ($2^{16}$ = 64KB)
    - A Physical Address (PA) is 12 bits ($2^{12}$ = 4KB)
    - The number of bits for the offset is 8 ($2^8$ = 256 bytes)

    The breakdown for a Virtual Address (VA) must be:

    - VA[15:14] (2 bits) : Denotes the region of memory (P0, P1, System)
    - VA[13:8] (6 bits) : The Virtual Page Number (VPN)
    - VA[7:0] (8 bits) : The offset

    The breakdown for a Physical Address (PA) must be:

    - PA[11:8] (4 bits) : The Page Frame Number (PFN)
    - PA[7:0] (8 bits) : The offset

    a. Answer: `x825C`

       Given the VAX 2-level translation scheme, we know that this VA must be in system space. Therefore, the top 2 bits of the VA (VA[15:14]) must be `10` (in binary). We also know that the offset of the VA is the same as the offset of the PA, so the bottom 8 bits of the VA (VA[7:0]) must be `x5C` (`01011100` in binary). Now, all we have to figure out is the 6 bit Virtual Page Number (VPN). It was given that the 1st access to physical memory (let's call this PA1) was at location `x108`. Once again, given the VAX 2-level translation scheme, we know that PA1 = SBR + (size of PTE in bytes) × VPN. Solving this equation for VPN we get VPN = (PA1 - SBR) ÷ (size of PTE in bytes).

       It was given that PA1 is `x108`, SBR is `x100`, and the size of a PTE is 4 bytes. Therefore, the VPN is `x2` (`000010` in binary). The complete VA is therefore `10 000010 01011100` (in binary) which is `x825C`.

    b. Answer: `x80000009`

       The contents of physical address `x45C` (the 2nd access to physical memory) is a PTE. We know that a PTE is a 32 bit value that consists of 1 valid bit (PTE[31]), and a 4-bit PFN (PTE[3:0]). All other bits of the PTE (PTE[30:4]) are 0. The contents of this PTE are used to form the address of the 3rd

access to physical memory (x942). Therefore, the PFN bits of the PTE must be x9, and the valid bit must be 1. This implies that the PTE is x80000009.

c. Answer x0342

First, we must determine if this VA is in P0 space or P1 space. To determine this, we have to figure out if P0BR or P1BR was used to compute the virtual address x825C (the answer to part a). It was given that P0BR is x8250, and that P1BR is x8350. Since x8350 is greater than x825C, we know that we could not have used P1BR to compute x825C, and therefore we must have used P0BR which means the VA is in P0 space. Therefore, the top 2 bits of the virtual address (VA[15:14]) must be 00 (in binary). We also know that the offset of the VA is the same as the offset of the PA, so the bottom 8 bits of the VA (VA[7:0]) must be x42 (01000010 in binary). Now, all we have to figure out is the 6 bit Virtual Page Number (VPN). Once again, given the VAX 2-level translation scheme, we know that x825C = P0BR + ((size of PTE in bytes) × VPN ).

Solving this equation for VPN we get VPN = (x825C - P0BR) ÷ (size of PTE in bytes).

It was given that P0BR is x8250, and the size of a PTE is 4 bytes. Therefore, the VPN is x3 (000011 in binary). The complete VA is therefore 00 000011 01000010 (in binary) which is x0342.

11. a. # virtual pages = virtual address space ÷ size of page = $2^{12}$ Bytes ÷ $2^5$ Bytes/page = $2^7$ pages

b. # physical frames = physical address space ÷ size of frame = $2^9$Bytes ÷ $2^5$ Bytes/frame = $2^4$ frames. Therefore, 4 bits are needed to specify the PFN.

Size of PTE = Valid bit + Modified bit + access control bits + PFN bits = 1 + 1 + 2 + 4 = 8 bits (1 Byte)

c. User space = (3/4) × Virtual address space = (3/4) × $2^7$ pages = 3 × $2^5$ pages. Each page of user space will have a PTE in the user space page table.

Size of user page table is # of entries × size of PTE = (3 × $2^5$ entries) × 1 Byte/entry = 96 Bytes.

# of pages = 96 Bytes ÷ $2^5$ Bytes/page = 3 pages.

d. We'll use the prefix “b” to indicate a binary number in this solution. Also, for clarity, we will call the virtual address x7AC the virtual address of *X* (VA_X).

Virtual address VA_X = x7AC
The three parts of this virtual address are:
VA_X[11:10]: b01 (indicates that this is an address in user space)
VA_X[11:5] (7 bits): Virtual Page Number = b0111101
VA_X[4:0] Offset within page: b01100

- *X* is on page x03D of user space
- VA of the PTE of the page containing x is VA_PTE_X = PTBR + (x03D × 1) = x380 + x03D = x3BD.
- Virtual page of System Space of PTE is VA_PTE_X[11:5] = x01D.
- PA of the PTE of this page of System Space is PA_PTE_PTE = SBR + VA_PTE_X[11:5] × 1 = x1E0 + x01D = x1FD.
- The PTE of this page of System Space is:
  PTE_PTE_X = Memory[x1FD] = xB8
  PFN_PTE_X = PTE_PTE_X[3:0] = x8
  PA of the PTE of the page containing *X*:
  PA_PTE_X = PFN_PTE_X concatenated with VA_PTE_X[4:0] = x11D

PTE_X = Memory[x11D] = x83
PFN_X = PTE_X[3:0] = x3
PA_X = PFN_X concantenated with VA_X[4:0] = x06C

12. We'll use the prefix “b” to indicate a binary number in this solution.

    Virtual address VA_X = x3456789A
    The three parts of this virtual address are:
    VA_X[31:30]: b00 (indicates that this is an address in P0 space)
    VA_X[29:9] (21 bits): Virtual Page Number = b 11 0100 0101 0110 0111 100 (x1A2B3C)
    VA_X[8:0] Offset within page: b010011010
    - x is on page x1A2B3C of P0 space
    - VA of the PTE of the page containing *X*:
      VA_PTE_X = P0BR + (x1A2B3C × 4) = x8AC40000 + x68ACF0 = x8B2CACF0
    - Virtual page of System Space of PTE is VA_PTE_X[29:9] = x59656
    - PA of the PTE of this page of System Space is PA_PTE_PTE = SBR + VA_PTE_X[29:9] × 4 = x22D958
    - The PTE of this page of System Space is PTE_PTE_X = Memory[x22D958] = x800F5D37
      PFN_PTE_X = PTE_PTE_X[20:0] = xF5D37
      PA of the PTE of the page containing *X*:
      PA_PTE_X = PFN_PTE_X concatenated with VA_PTE_X[8:0] = x1EBA6EF0
      PTE_X = Memory[x1EBA6EF0] = x80000A72
      PFN_X = PTE_X[20:0] = xA72
      PA_X = PFN_X concantenated with VA_X[8:0] = x14E49A

13. Including instruction fetch, every instruction can generate a page fault. Ignoring instruction fetch, LDB, LDW, STB, STW, TRAP, RTI can generate a page fault (If the trap vector table or system stack is always in physical memory, then the TRAP or RTI won't generate a page fault).

    Including the instruction fetch, RTI can generate the maximum number of page faults (3) and LDB, LDW, STB, STW, TRAP can generate the next most number of page faults (2). (Points will not be deducted for RTI)

14. This problem has been postponed to later problem sets

15. This problem has been postponed to later problem sets

16. This problem has been postponed to later problem sets

17. This problem has been postponed to later problem sets

18. This problem has been postponed to later problem sets