Zak Rowland
11/14/2019
CST337
Lab 5 answers

Part 1:

- 210ns = 4.762MHz
  - 4.762MHz = $\frac{84MHz}{2x+2}$
  - (2x+2) 4.762MHz = 84MHz
  - 9.5238x MHz + 9.5238MHz = 84MHz
  - 9.5238x MHz = 74.477MHz
  - x = 7.82
- BRG = 8 -> SPICLK = 4.66666MHz -> 214.28ns
  - Actual measured SPICLK = 216ns = 4.63MHz

Parameter 3 requirement:

```
void WriteEnable(void)
{
    LATFCLR = _LATF_LATF8_MASK; //Assert CS (LATF8)
    SPI4BUF = SPI_WREN; //Write a write enable command to the 25LC256
    while(SPI4STATbits.SPIRBF == 0){} //Wait for Receive Buffer Full (RBF = 1)
    SPI4BUF; //Read and discard the dummy data that was clocked in
    LATFSET = _LATF_LATF8_MASK; //Negate CS
}
!void WriteEnable(void)
!{
0x9D000618: ADDIU SP, SP, -8
0x9D00061C: SW FP, 4(SP)
0x9D000620: ADDU FP, SP, ZERO
!    LATFCLR = _LATF_LATF8_MASK; //Assert CS (LATF8)
0x9D000624: LUI V0, -16506
0x9D000628: ADDIU V1, ZERO, 256
0x9D00062C: SW V1, 1332(V0)
!    SPI4BUF = SPI_WREN; //Write a write enable command to the 25LC256
0x9D000630: LUI V0, -16510
0x9D000634: ADDIU V1, ZERO, 6
0x9D000638: SW V1, 5664(V0)
!    while(SPI4STATbits.SPIRBF == 0){} //Wait for Receive Buffer Full (RBF = 1)
0x9D00063C: NOP
0x9D000640: LUI V0, -16510
0x9D000644: LW V0, 5648(V0)
0x9D000648: ANDI V0, V0, 1
0x9D00064C: BEQ V0, ZERO, 0x9D000640
0x9D000650: NOP
!    SPI4BUF; //Read and discard the dummy data that was clocked in
0x9D000654: LUI V0, -16510
0x9D000658: LW V0, 5664(V0)
!    LATFSET = _LATF_LATF8_MASK; //Negate CS
0x9D00065C: LUI V0, -16506
0x9D000660: ADDIU V1, ZERO, 256
0x9D000664: SW V1, 1336(V0)
!}
0x9D000668: ADDU SP, FP, ZERO
0x9D00066C: LW FP, 4(SP)
0x9D000670: ADDIU SP, SP, 8
0x9D000674: JR RA
0x9D000678: NOP
```

Zak Rowland
11/14/2019
CST337
Lab 5 answers

The worst cases are WriteEnable() and Write() which both have 9 machine instructions between RBF high and CS high.

$\frac{200ns-\left(\frac{216ns}{2}\right)}{11.9ns}$ = 7.73 instructions, or 8 instructions required in the worst case. Since I have 9 instructions in my worst case, I meet the parameter 3 requirement.

Parameter 4 requirement:

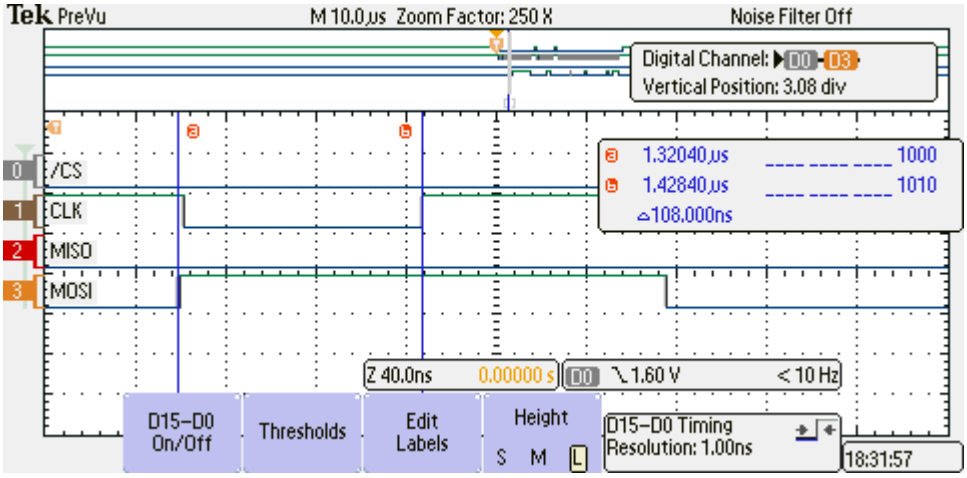
```
void WriteEnable(void)
{
    LATFCLR = _LATF_LATF8_MASK; //Assert CS (LATF8)
    SPI4BUF = SPI_WREN; //Write a write enable command to the 25LC256
    while(SPI4STATbits.SPIRBF == 0){} //Wait for Receive Buffer Full (RBF = 1)
    SPI4BUF; //Read and discard the dummy data that was clocked in
    LATFSET = _LATF_LATF8_MASK; //Negate CS
}

!void WriteEnable(void)
!{
    .
    .
    .
!    LATFSET = _LATF_LATF8_MASK; //Negate CS
0x9D00065C: LUI V0, -16506
0x9D000660: ADDIU V1, ZERO, 256
0x9D000664: SW V1, 1336(V0)
!}
0x9D000668: ADDU SP, FP, ZERO
0x9D00066C: LW FP, 4(SP)
0x9D000670: ADDIU SP, SP, 8
0x9D000674: JR RA
0x9D000678: NOP

void Write(uint16_t base_address, uint8_t num_of_bytes, uint8_t * data)
{
    uint16_t i = 0;
    while(ReadStatus() & 0x01){} //Wait for write in progress
    WriteEnable();
    LATFCLR = _LATF_LATF8_MASK; //Assert CS (LATF8)
    .
    .
    .
}

!void Write(uint16_t base_address, uint8_t num_of_bytes, uint8_t * data)
!{
    .
    .
    .
!    WriteEnable();
0x9D000720: JAL WriteEnable
0x9D000724: NOP
!    LATFCLR = _LATF_LATF8_MASK; //Assert CS (LATF8)
0x9D000728: LUI V0, -16506
0x9D00072C: ADDIU V1, ZERO, 256
0x9D000730: SW V1, 1332(V0)
    .
    .
    .
!}
```

Zak Rowland
11/14/2019
CST337
Lab 5 answers

The worst case where CS is negated then quickly asserted again is the negation of CS in the WriteEnable() function followed by assertion of CS in the Write() function. There are 0 C instructions between these points, however since WriteEnable() is a function call there are 7 SYSCLK cycles between the SW of the negation and the SW of the assertion. This means the parameter 4 requirement is met without adding NOPs.

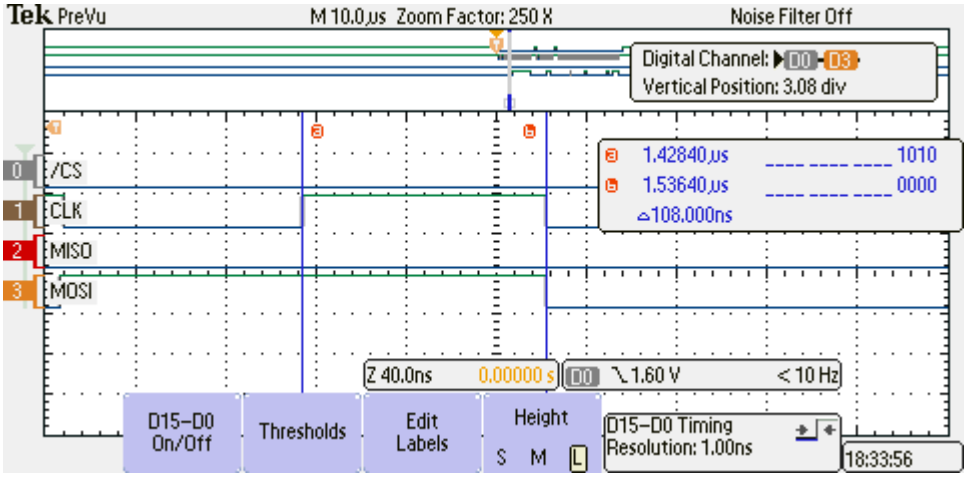
Parameter 5 – LC256 setup: MET

- Required: 20ns – Measured: 108ns +/- 1ns



Parameter 6 – LC256 hold: MET

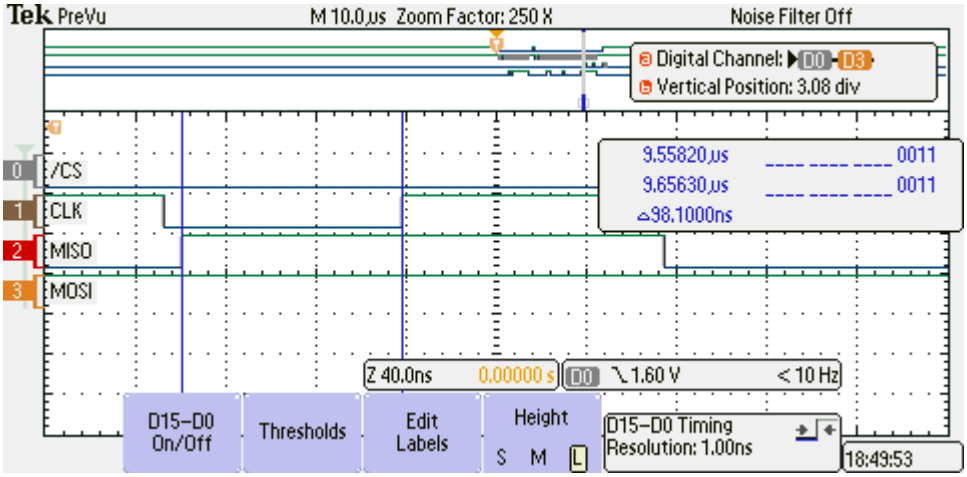
- Required: 40ns – Measured: 108ns +/- 1ns



Zak Rowland
11/14/2019
CST337
Lab 5 answers

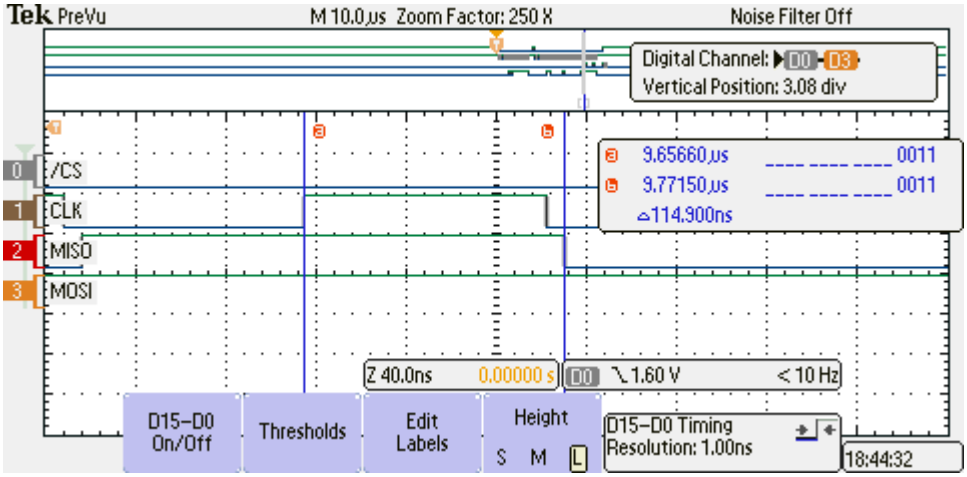
Parameter 40 – PIC32MZ setup: MET

- Required: 7ns – Measured: 98.1ns +/- 1ns



Parameter 41 – PIC32MZ hold: MET

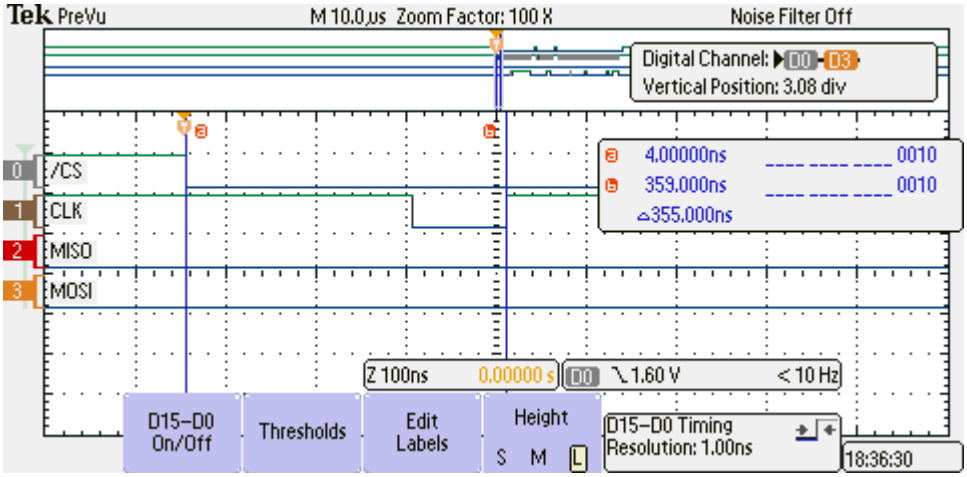
- Required: 7ns – Measured: 114.9ns +/- 1ns



Zak Rowland
11/14/2019
CST337
Lab 5 answers

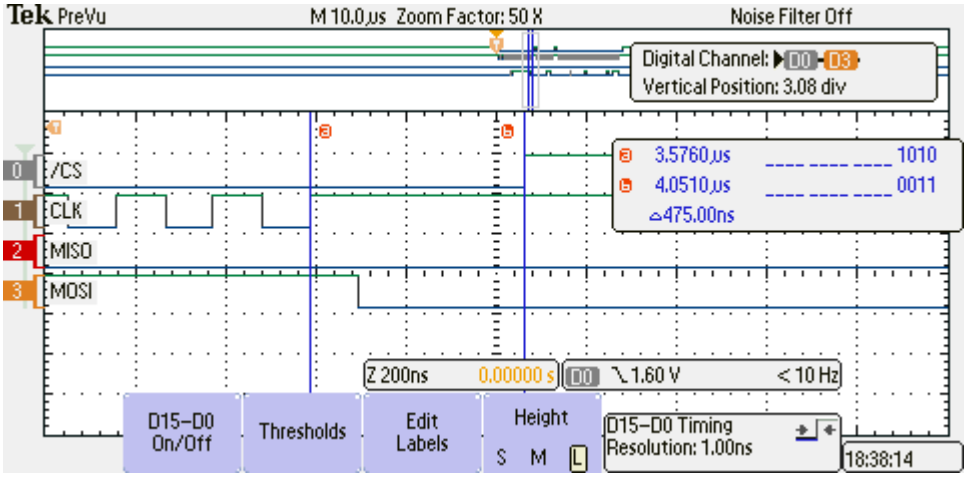
Parameter 2– LC256 CS setup: MET

- Required: 100ns – Measured: 355ns +/- 1ns



Parameter 3 – LC256 CS hold: MET

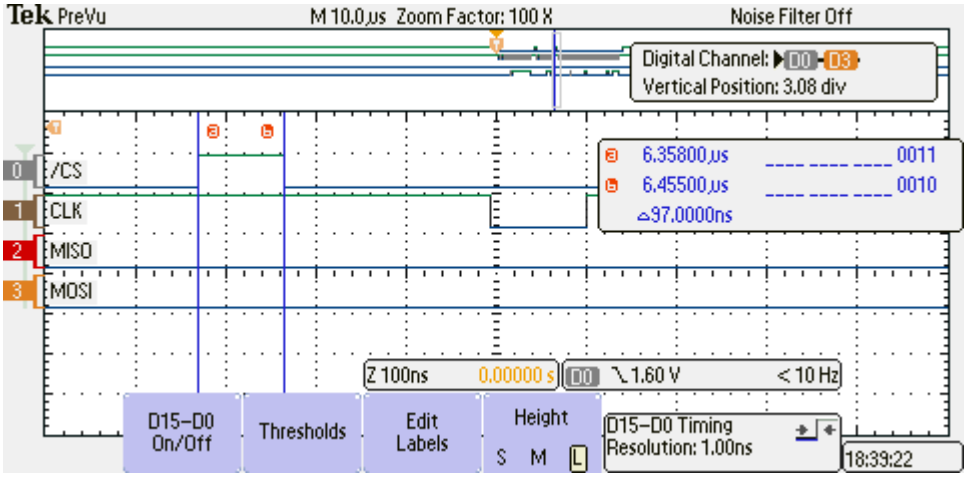
- Required: 200ns – Measured: 475ns +/- 1ns



Zak Rowland
11/14/2019
CST337
Lab 5 answers

Parameter 4 – LC256 CS disable: MET

- Required: 50ns – Measured: 97ns +/- 1ns



System clocks between write to empty SPI4BUF and when TBE is set:

By using the built in stopwatch and setting breakpoints on the wait for TBE instruction and the instruction immediately after, I get a range of 22-48 cycles. I believe the stopwatch not being entirely accurate causes the uncertainty.

Part 2:

Parameter 4 requirement:

```
void __ISR_AT_VECTOR(_SPI4_RX_VECTOR, IPL7SRS) SpiServ(void)
{
    uint8_t returned_data = 0;
    switch(EEPromState)
    {
        .
        .
        .
        case 3: //End write enable, start write sequence and write MSA
            SPI4BUF; //Read and discard the dummy data that was clocked in
            LATFSET = _LATF_LATF8_MASK; //Negate CS
            LATFCLR = _LATF_LATF8_MASK; //Assert CS (LATF8) !HERE!
            SPI4BUF = SPI_WRITE; //Write a write command to the 25LC256
            while(SPI4STATbits.SPITBE == 0){} //Wait for Transmit Buffer Empty (TBE = 1)
            SPI4BUF = EEPromAddress >> 8; //Write MSA to the 25LC256
            EEPromState = 4; //Set state to 4
            break;

!           LATFSET = _LATF_LATF8_MASK; //Negate CS
0x9D0003F0: LUI V0, -16506
0x9D0003F4: ADDIU V1, ZERO, 256
0x9D0003F8: SW V1, 1336(V0)
!            LATFCLR = _LATF_LATF8_MASK; //Assert CS (LATF8) !HERE!
0x9D0003FC: LUI V0, -16506
0x9D000400: ADDIU V1, ZERO, 256
0x9D000404: SW V1, 1332(V0)
```

Zak Rowland
11/14/2019
CST337
Lab 5 answers

The worst case where CS is negated then quickly asserted again is in state 3 of the state machine which is the write enable end and start write state. There are 0 C instructions between these points which means there are only two SYSCLK cycles. I had to add three NOPs to meet the parameter 4 requirement of at least five SYSCLK cycles.

Parameter 2 – LC256 CS setup: MET

- Required: 100ns – Measured: 376.8ns +/- 1ns

Parameter 3 – LC256 CS hold: MET

- Required: 200ns – Measured: 696.8ns +/- 1ns

Parameter 4 – LC256 CS disable: MET

- Required: 50ns – Measured: 168ns +/- 1ns

Minimum clock period used for operation: 214.29ns

- ~110 cycles between LED toggles
- 214.29ns * 110 cycles = 23.57us between interrupts for a read sequence

Latency+execution times for each state of a two-byte read sequence:

- State 0 (start check status): 1.38us +/- 1ns
- State 1 (mid check status): 752ns +/- 1ns
- State 2 (end check, start read with MSA): 1.424us +/- 1ns
- State 7 (write LSA): 784ns +/- 1ns
- State 8 (read first byte): 1.07us +/- 1ns
- State 9 (store first byte, read second byte): 1.33us +/- 1ns
- State 9 (store second byte, switch state): 1.53us +/- 1ns
- State 10 (read sequence done): 816ns +/- 1ns

Depending on the state, there is about 21 microseconds of time remaining to execute other code between interrupts. 21us is enough time for some C instructions such as toggling an LED, but it is not much. To see a greater advantage over the polling method the SPI clock would have to run faster.

Zak Rowland
11/14/2019
CST337
Lab 5 answers

Service routine description:

The state machine is comprised of 11 total states (0-10.) Regardless of whether read or write is chosen, the machine will always start in state 0. State 0 begins the check status sequence by asserting CS. The read status command is written to the 25LC256, then it waits until the transmit buffer is empty. Once the transmit buffer is empty, a byte of dummy data is written to the 25LC256 and the state is set to 1. State 1 is the middle of the check status sequence, and it simply reads the dummy byte that was clocked in from sending the read status command in state 0. After reading the dummy data, the state is set to 2.

State 2 finishes the check status sequence and either repeats the check status sequence or begins a read or write sequence. This state begins by storing the status byte that was clocked in while writing a dummy byte in state 0 then negating CS. The stored status byte is then AND with 0x01 to determine if there is a write in progress, and if there is then the check status sequence begins again (same as state 0.) If a write is not in progress, it is safe to start a read or write sequence. If a write is chosen, the sequence begins by initializing the write counter to 0 and asserting CS. The write enable command is then written to the 25LC256, and after the transmit buffer is empty the state is set to 3. If a read is chosen, the sequence begins by initializing the read counter to 0, a temporary read variable to 0, and asserting CS. The read command is written to the 25LC256, and after the transmit buffer is empty the MSA is written to the 25LC256. After the MSA is written, the state is set to 7.

State 3 continues the write sequence by first finishing the write enable. This state begins by reading a dummy byte that was clocked in from sending the write enable command in state 2, then CS is negated. After the three NOPs required to meet timing requirements, CS is asserted again and the write command is written to the 25LC256. Once the transmit buffer is empty, the MSA is written and the state is set to 4.

State 4 continues the write sequence by first finishing the MSA write. This state begins by reading a dummy byte that was clocked in from sending the MSA in state 3. The LSA is then written to the 25LC256 and the state is set to 5.

Zak Rowland
11/14/2019
CST337
Lab 5 answers

State 5 continues the write sequence by first finishing the LSA write. This state begins by reading a dummy byte that was clocked in from sending the LSA in state 4. If the write counter is less than the number of bytes needed to write, continue with writing a byte to the 25LC256 from the write buffer. Once a byte is written, increment the counter and remain in state 5. Repeat state 5 until the counter is not less than the number of bytes need to write, then the state is set to 6.

State 6 is the last state in the write sequence. This state begins by reading a dummy byte that was clocked in from sending the last byte of data in state 5. After reading the dummy byte, CS is negated and the state machine is reset by setting the busy flag to 0 and the state 0.

State 7 continues the read sequence by first finishing the MSA write. This state begins by reading a dummy byte that was clocked in from sending the MSA in state 3. The LSA is then written to the 25LC256 and the state is set to 8.

State 8 continues the read sequence by first finishing the LSA write. This state begins by checking if the read counter is less than the number of bytes needed to read + 1. If not, move to state 10. If so, it starts by temporarily storing the data clocked in and writing a dummy byte to the 25LC256. This is because it must output a dummy byte for every data byte needed to read. After writing the dummy byte, increment the read counter and set the state to 9.

State 9 continues the read sequence. This state begins by storing the clocked in data to the read buffer and checking if the read counter is less than the number of bytes needed to read + 1. If not, move to state 10. If so, it starts by temporarily storing the data clocked in and writing a dummy byte to the 25LC256 if the counter is less than the number of bytes needed to read. If the read counter is greater than 1, store the temporary data into the read buffer then increment the read buffer pointer. Finally, increment the read counter and if it still less than the number of bytes needed to be read + 1, then stay in state 9 and repeat until all bytes are read. If not, set the state to 10.

State 10 is the last state in the read sequence. This state begins by storing the data that was clocked in into the read buffer. Once this data is stored, CS is negated and the state machine is reset by setting the busy flag to 0 and the state to 0.

After running through any state, the SPI4RX interrupt flag must be cleared to allow the next interrupt to trigger.

Zak Rowland
11/14/2019
CST337
Lab 5 answers