ECE3849
D-Term 2021

Real Time Embedded Systems

Module 5 Part 2

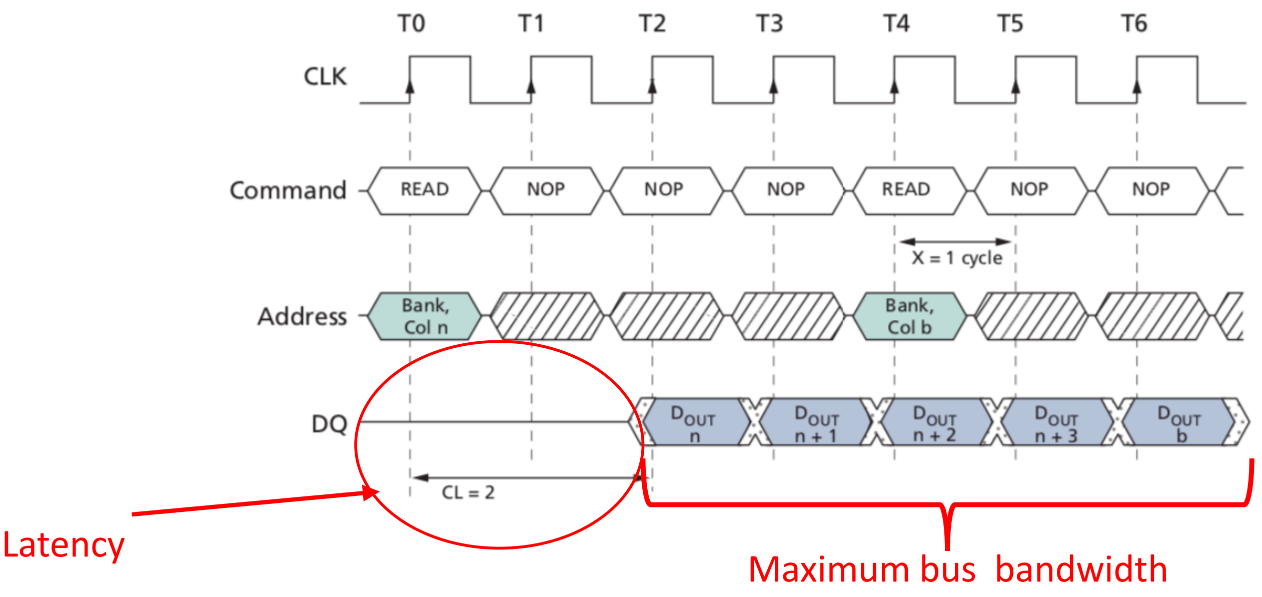
Module 5 Part 2 Overview

- Bus utilization and burst transfers.
- Benefits and performance of FIFOs.
- ADC FIFO Example.
- Direct Memory Access.

Bursting Transactions

- To maximize bus bandwidth, it is important to remove the effect of latency.
- This is done by issuing consecutive bursts of transactions without waiting for the previous transactions to complete.
  - The types of allowable burst transactions depend on the specific memory or peripheral you are using.
  - Bursting multiple consecutive reads and writes are the most common transactions.
  - Below is the bus timing for multiple consecutive reads on an SDRAM device.
    - The read data takes several clock cycles before it is available (latency).
    - But once the reads are started, a new data value comes out every clock cycle. This allows for the system's maximum bus bandwidth to be realized during the burst period.

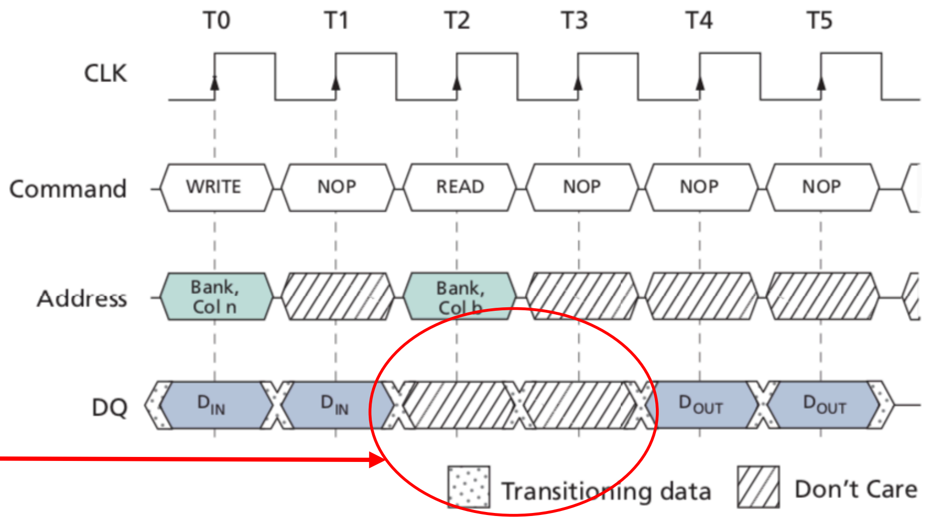
Consecutive READ Bursts



Non-burst transaction

- Often there are transactions where maximum performance can not be achieved and latency reduces bus performance.
  - Switching between reads and writes on a bus with bi-directional data bus.
  - Performing transactions to different peripherals, you often need fully finish a transaction on one peripheral before accessing a new one.
  - The example below shows a write-to-read transaction on an SDRAM,
    - 2 of the 6 clock cycles are not transferring data.
    - The bandwidth of the bus is only achieving 66% of its maximum bandwidth due to the latency of the read operation. Bus utilization is 66%.
- Bus utilization is the percentage of bus cycles used for transferring bus transactions of any type. In this example bus utilization 66%

WRITE-to-READ



Dead time caused by the write command needing to finish before the read command could start.

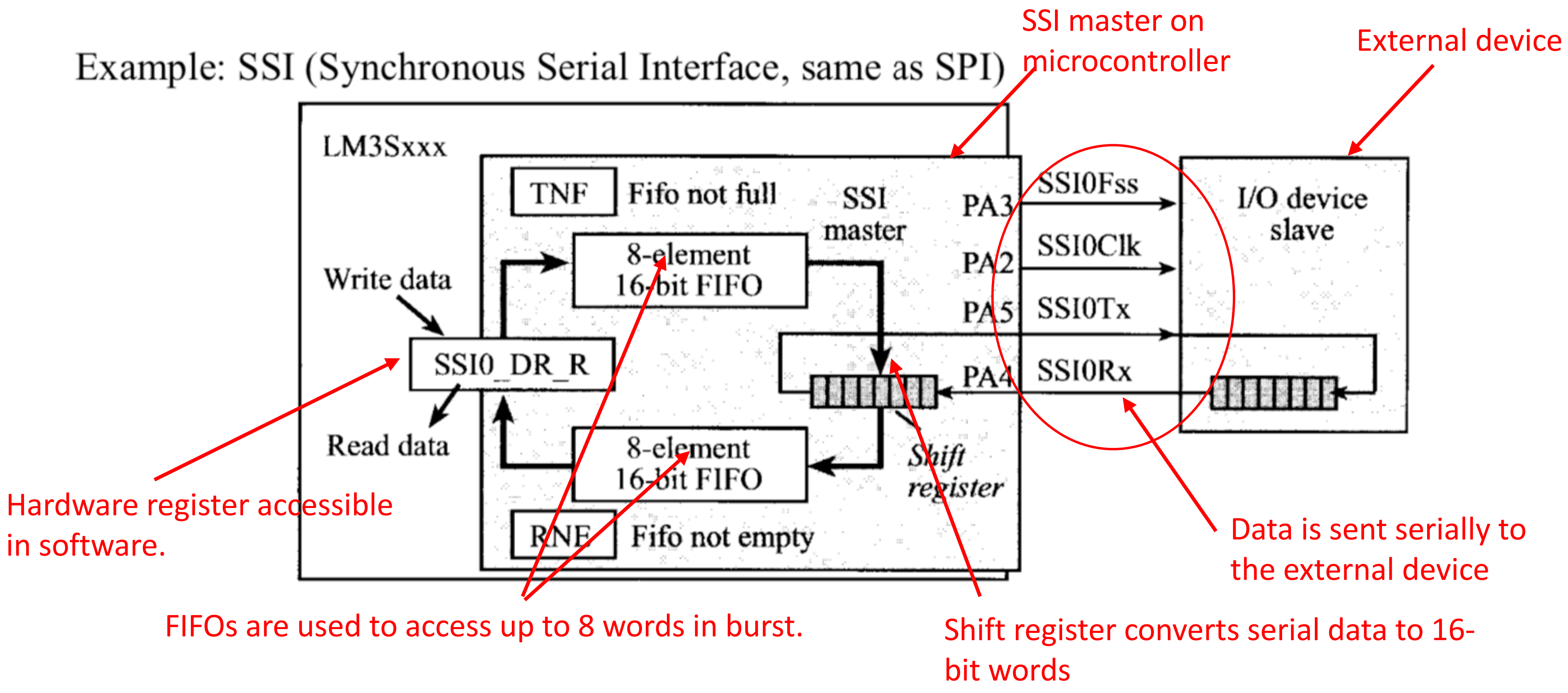
Benefits of FIFOs

- Because of the benefit of burst transactions on bus utilization, many peripherals employ FIFOs.
  - Without FIFOs, one piece of information would need to be accessed at a time. The latency between each access reduces bus utilization.
  - FIFOs allow bursts of data to be sent at a time, increasing bus utilization and improving maximum bus bandwidth.
- FIFOs can relax the relative deadlines for your system's ISR.
  - For example, in the labs our ADC_ISR needs to be called every 1 usec. Once for every single ADC acquisition.
  - If we use the FIFO in the ADC module, we could receive 4 samples at a time. The period for the ISR is now increased to 4 usec.
- Reducing the number of interrupt calls and the efficiency of bursting transactions, lowers CPU load.

SSI FIFO Example

- Without FIFOs, each access to the peripheral would need to wait for each serial transaction to the external device to complete before starting a second transaction.
  - Depending on the clock rate of SSI0Clk, this could be a very long time.
- FIFO's are added to the read and write data paths.
  - This allows the CPU to burst up to 8 16-bit words at a time.
  - This is accomplished by performing multiple consecutive reads and writes to the SSI0_DR_R hardware register.
  - The CPU no longer needs to wait for each serial transaction on the bus to complete.

Example: SSI (Synchronous Serial Interface, same as SPI)

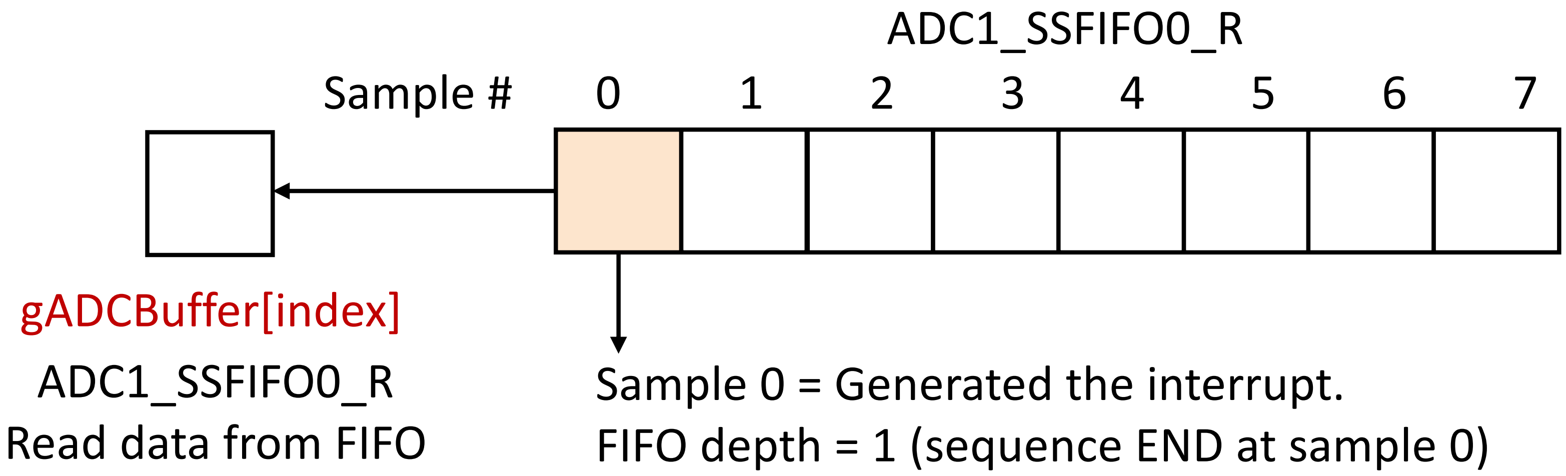

SSI master on microcontroller
External device
LM3Sxxx
TNF
Fifo not full
SSI master
8-element 16-bit FIFO
Write data
SSI0_DR_R
Read data
8-element 16-bit FIFO
RNF
Fifo not empty
Shift register
PA3
PA2
PA5
PA4
SSI0Fss
SSI0Clk
SSI0Tx
SSI0Rx
I/O device slave
Hardware register accessible in software.
FIFOs are used to access up to 8 words in burst.
Shift register converts serial data to 16-bit words
Data is sent serially to the external device

ece3849_adc_fifo Example

- Each ADC sequencer has a FIFO.
  - For sequence 0, the FIFO can be up to 8 samples deep.
- ece3849_adc_fifo explores the effect of the ADC FIFO on
  - CPU load.
  - ISR Period Requirements.
  - Relative deadlines for reading the data.
- The program implements the following functionality
  - Initializes the sampling rate to 1 usec / 1 MHz.
  - Implements the CPU load calculation from previous examples.
  - Configures the ADC in different modes
    - Lab 1 & 2 setup
      - Single sample with a FIFO depth of 1.
    - FIFO depth of 8
      - With interrupts triggered every 2, 4, and 8 samples.
  - In the while() loop,
    - It performs the CPU load calculations.
    - It has the option of disabling interrupts for an adjustable number of usec to force increases in interrupt latency.

ADC Performance without FIFOs

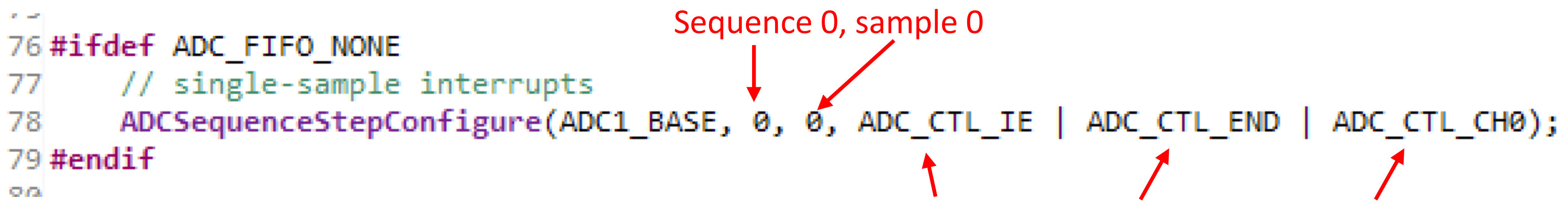
- In Lab 1 and 2, we configured our ADC to take only one sample per sequence and then trigger an interrupt.
  - If not read in time, an overflow condition occurred and data was lost.
- We needed to read the ADC value from ADC_ISC_IN0 on every single conversion.
  - The ADC_ISR period is 1 usec.
  - The measured CPU load was around 60%.
  - It required a “zero latency interrupt” to meet the deadline. Even the small amounts of latency added by the RTOS would cause it to break.
  - ADC_ISR maximum schedulable response time is 1 usec.



ece3849_adc_fifo / No FIFO

- ADC FIFO Configuration (No FIFO / Lab 1 & 2)

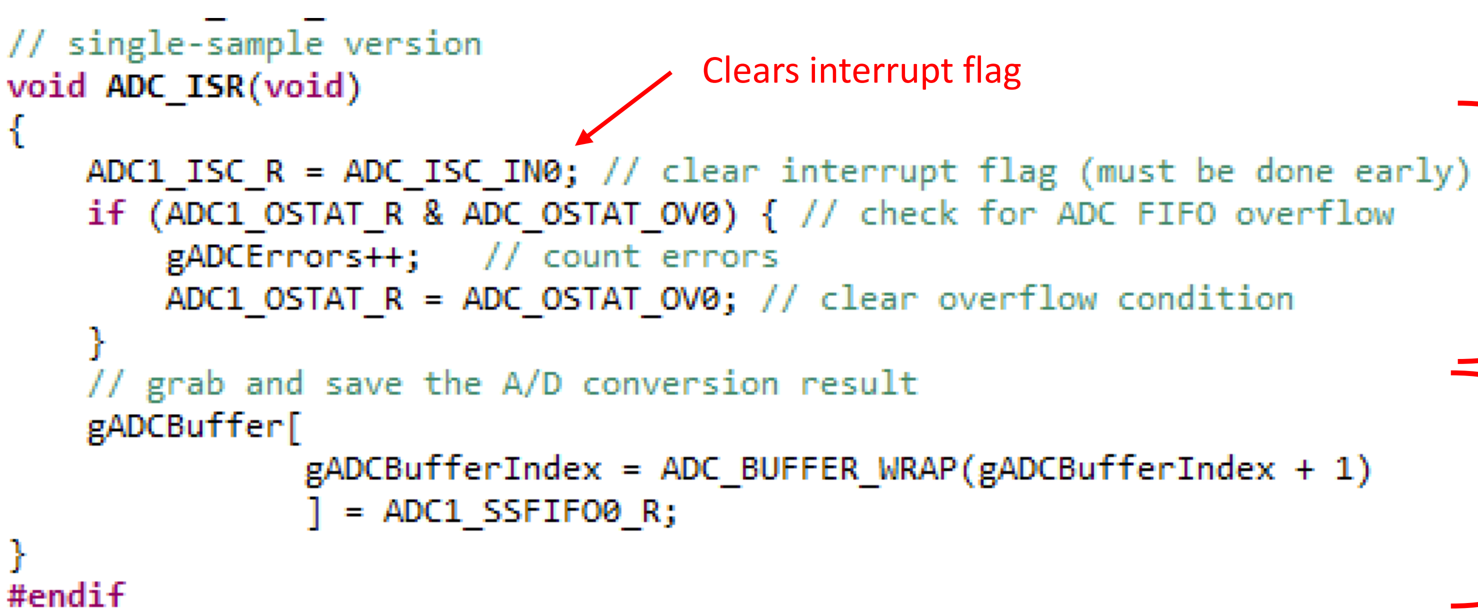
Sequence 0, sample 0



```
#ifdef ADC_FIFO_NONE
    // single-sample interrupts
    ADCSequenceStepConfigure(ADC1_BASE, 0, 0, ADC_CTL_IE | ADC_CTL_END | ADC_CTL_CH0);
#endif
```

Trigger an interrupt on first sample from CH0 and ends sequence

- ADC_ISR functionality



```
// single-sample version
void ADC_ISR(void)
{
    ADC1_ISC_R = ADC_ISC_IN0; // clear interrupt flag (must be done early)
    if (ADC1_OSTAT_R & ADC_OSTAT_OV0) { // check for ADC FIFO overflow
        gADCErrors++;    // count errors
        ADC1_OSTAT_R = ADC_OSTAT_OV0; // clear overflow condition
    }
    // grab and save the A/D conversion result
    gADCBuffer[
               gADCBufferIndex = ADC_BUFFER_WRAP(gADCBufferIndex + 1)
               ] = ADC1_SSFIFO0_R;
}
#endif
```

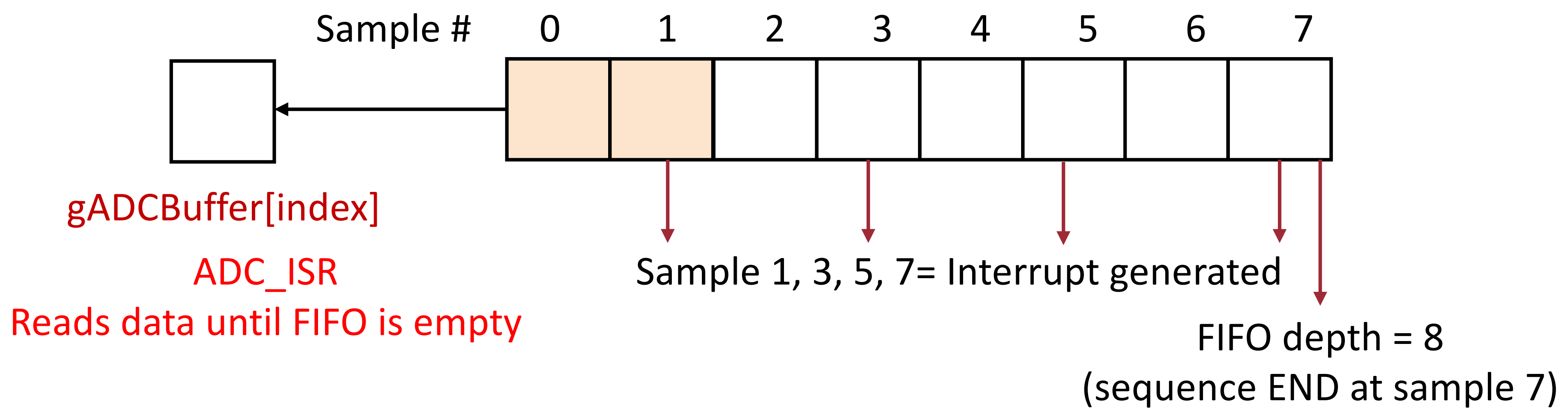
Clears interrupt flag

Checks for overflow condition and counts errors.

Reads one Sample.

Effects of ADC FIFO depth of 8

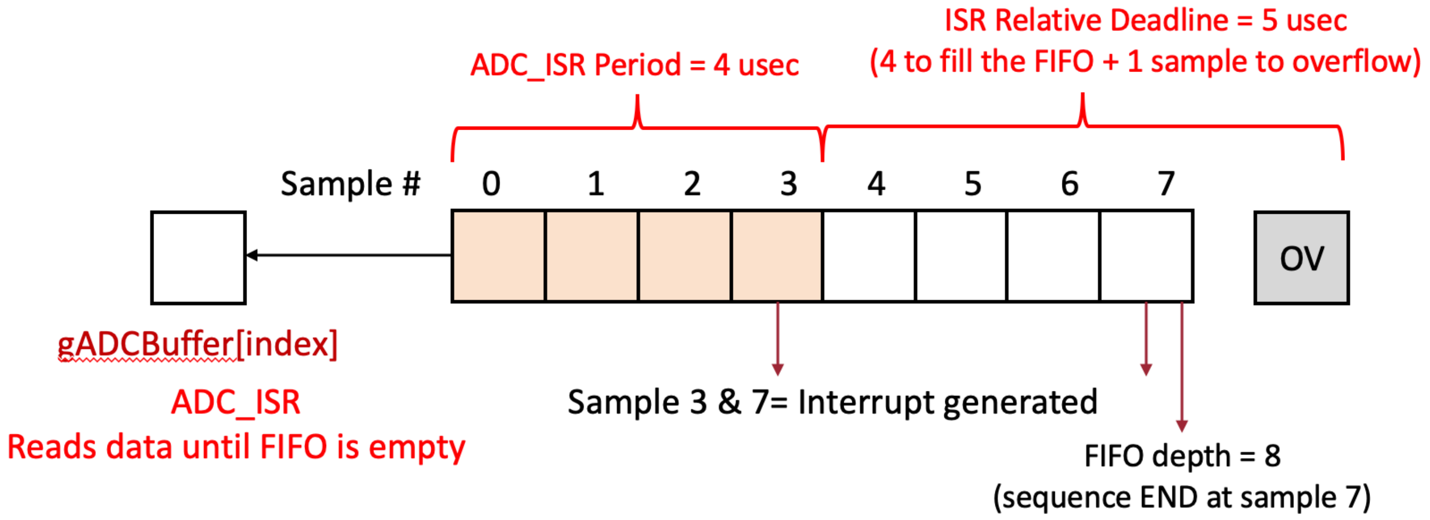
- With a FIFO depth of 8, we can read 8 samples before overflowing the FIFO and loosing data.
- We can also configure an interrupt at any point in the FIFO to read the data.
- If we configure an interrupt every 2 samples, the ISR will read the FIFO data until the FIFO is empty.
  - What is our new ISR period sampling at 1 MHz?
    - ?
  - After the ISR is triggered, how long before the FIFO is full?
    - ?
  - How long until we have an overflow condition? (relative deadline)
    - ?



Sample #
0
1
2
3
4
5
6
7
gADCBuffer[index]
ADC_ISR
Reads data until FIFO is empty
Sample 1, 3, 5, 7= Interrupt generated
FIFO depth = 8
(sequence END at sample 7)

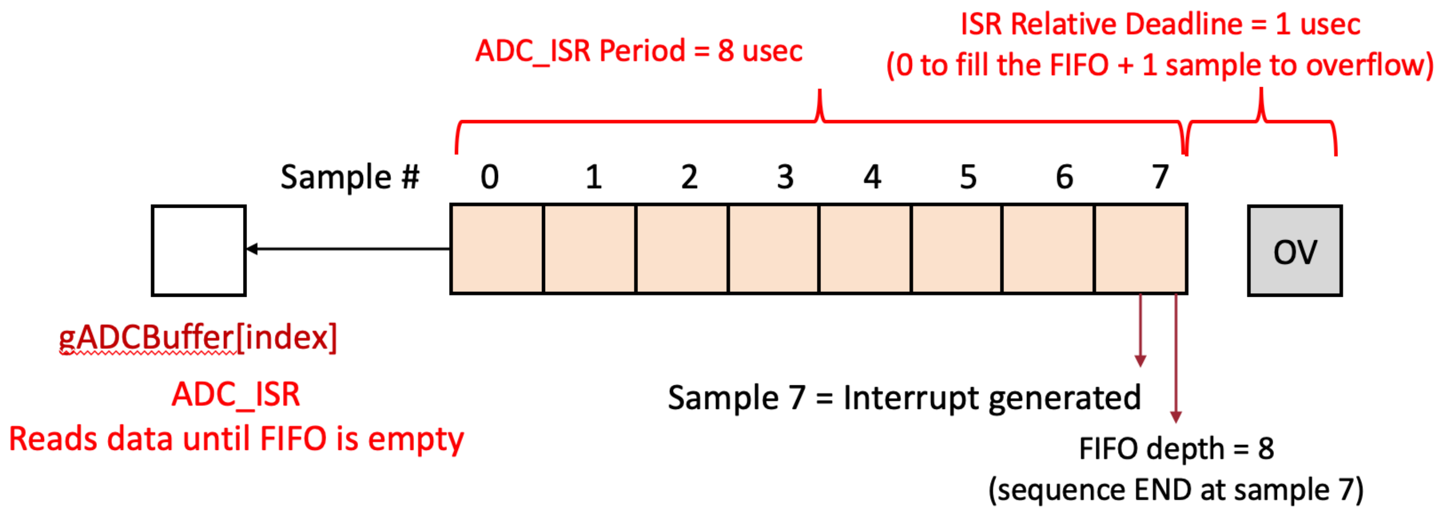
Effect of Interrupt Period

- Interrupts every 4 samples



ADC_ISR Period = 4 usec
ISR Relative Deadline = 5 usec
(4 to fill the FIFO + 1 sample to overflow)
Sample #
0
1
2
3
4
5
6
7
OV
gADCBuffer[index]
ADC_ISR
Reads data until FIFO is empty
Sample 3 & 7= Interrupt generated
FIFO depth = 8
(sequence END at sample 7)

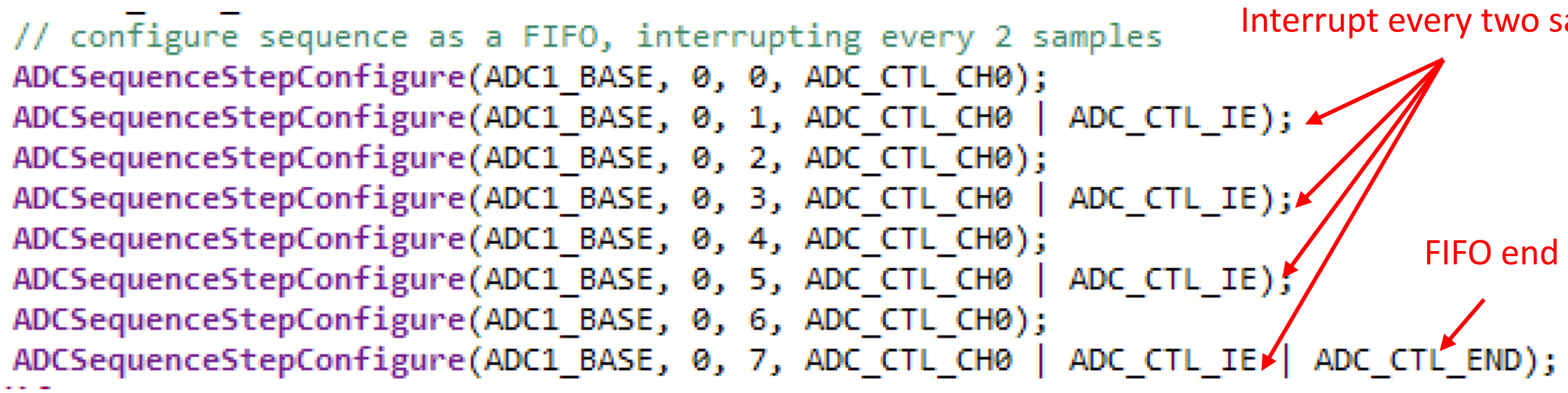
- Interrupts every 8 samples



ADC_ISR Period = 8 usec
ISR Relative Deadline = 1 usec
(0 to fill the FIFO + 1 sample to overflow)
Sample #
0
1
2
3
4
5
6
7
OV
gADCBuffer[index]
ADC_ISR
Reads data until FIFO is empty
Sample 7 = Interrupt generated
FIFO depth = 8
(sequence END at sample 7)

ece3849_adc_fifo /FIFO Size = 8

- ADC FIFO Configuration: FIFO Size = 8, Interrupt every 2 samples.

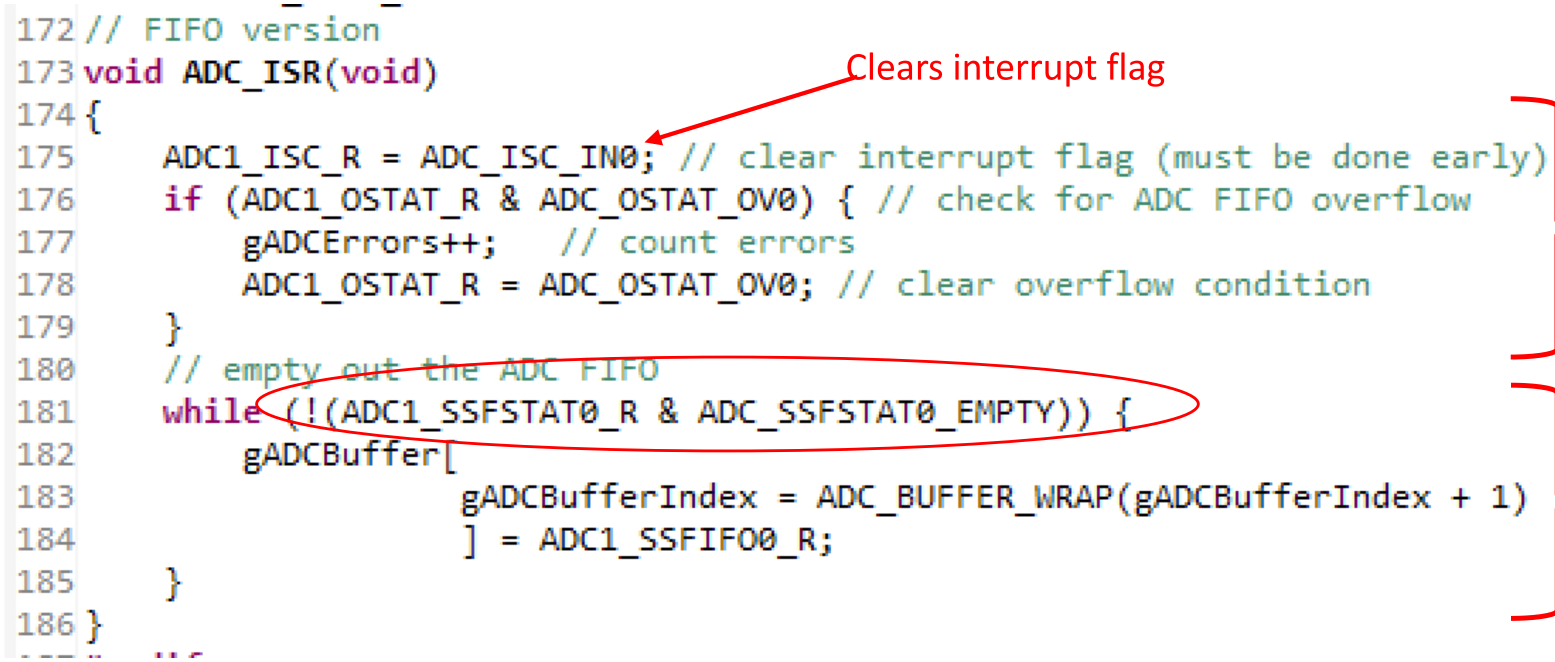


```
// configure sequence as a FIFO, interrupting every 2 samples
ADCSequenceStepConfigure(ADC1_BASE, 0, 0, ADC_CTL_CH0);
ADCSequenceStepConfigure(ADC1_BASE, 0, 1, ADC_CTL_CH0 | ADC_CTL_IE);
ADCSequenceStepConfigure(ADC1_BASE, 0, 2, ADC_CTL_CH0);
ADCSequenceStepConfigure(ADC1_BASE, 0, 3, ADC_CTL_CH0 | ADC_CTL_IE);
ADCSequenceStepConfigure(ADC1_BASE, 0, 4, ADC_CTL_CH0);
ADCSequenceStepConfigure(ADC1_BASE, 0, 5, ADC_CTL_CH0 | ADC_CTL_IE);
ADCSequenceStepConfigure(ADC1_BASE, 0, 6, ADC_CTL_CH0);
ADCSequenceStepConfigure(ADC1_BASE, 0, 7, ADC_CTL_CH0 | ADC_CTL_IE | ADC_CTL_END);
```

Interrupt every two samples

FIFO end at sample 7

- ADC_ISR functionality



```
// FIFO version
void ADC_ISR(void)
{
    ADC1_ISC_R = ADC_ISC_IN0; // clear interrupt flag (must be done early)
    if (ADC1_OSTAT_R & ADC_OSTAT_OV0) { // check for ADC FIFO overflow
        gADCErrors++;   // count errors
        ADC1_OSTAT_R = ADC_OSTAT_OV0; // clear overflow condition
    }
    // empty out the ADC FIFO
    while (!(ADC1_SSFSTAT0_R & ADC_SSFSTAT0_EMPTY)) {
        gADCBuffer[
                   gADCBufferIndex = ADC_BUFFER_WRAP(gADCBufferIndex + 1)
                   ] = ADC1_SSFIFO0_R;
    }
}
```

Clears interrupt flag

Checks for overflow condition and counts errors.

Reads until the FIFO is empty.

ece3849_adc_fifo / Results

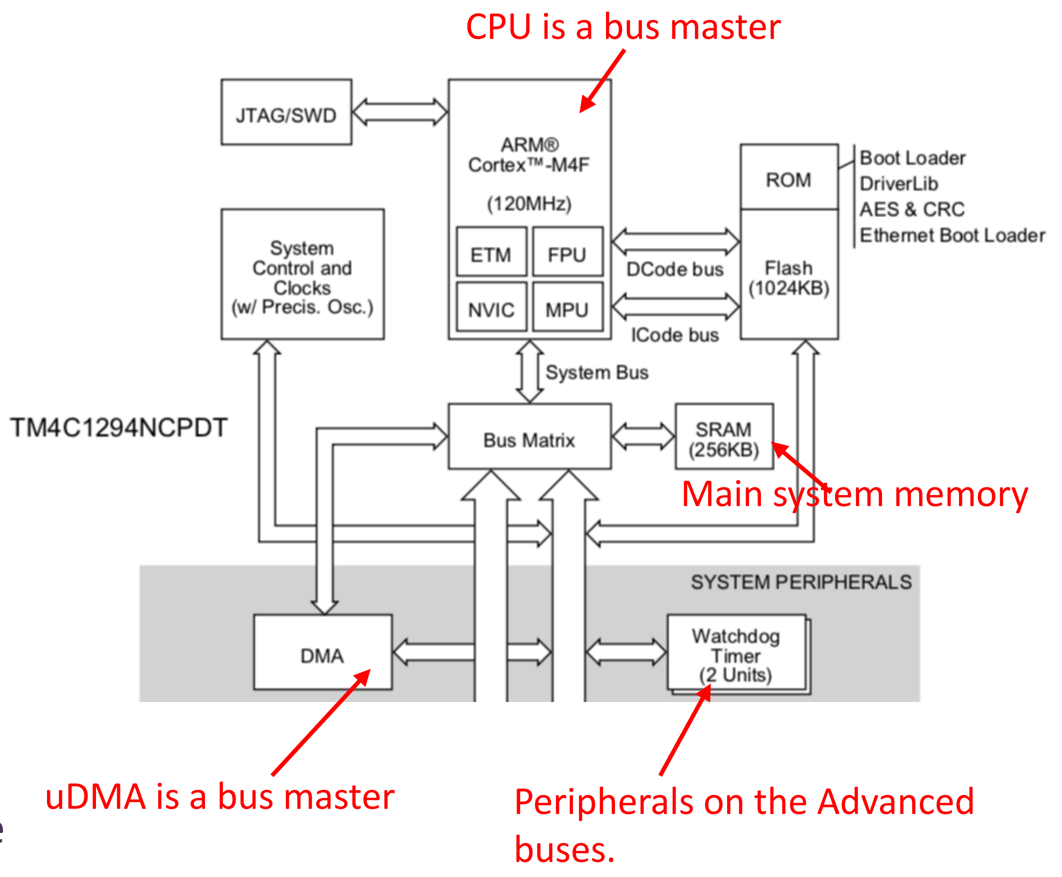
- Performance trade-offs
  - Single Sample versus FIFO
    - Using a FIFO increases the required ISR Period (lower rate) thus lowering the CPU load.
  - Once using a FIFO
    - Lower ISR periods (higher rate), increases the CPU load but allows for longer relative deadlines.
    - Higher ISR periods (lower rate), reduces the CPU load but shortens the relative deadlines requiring tighter schedules with less margin.

ADC FIFO Configuration	ISR Period	Relative Deadline	CPU Load
Single Sample	1 usec	1 usec	62%
FIFO (size = 8), Interrupts every 2 samples	2 usec	7 usec	43%
FIFO (size = 8), Interrupts every 4 samples	4 usec	5 usec	28%
FIFO (size = 8), Interrupts every 8 samples	8 usec	1 usec	20%

DMA (Direct Memory Access)

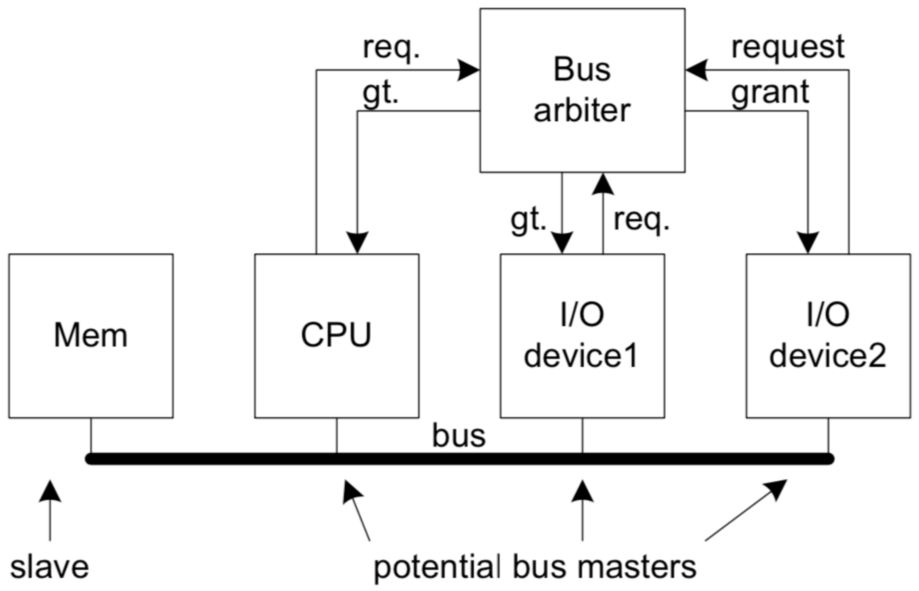
- Direct Memory Access (DMA) functionality allows reading and writing to main system memory independent of the CPU.
  - We no longer need to use CPU cycles to transfer data to the SRAM.
  - Peripherals can read / write directly to the memory using the uDMA controller on the TM4C1294 controller.

- To allow direct transfer the uDMA controller is a bus master.
- There are two masters on the bus matrix, the uDMA controller and the CPU.
- The advantages of DMA
  - Lower CPU load
    - Fewer ISRs.
    - Fewer data transfers in software.
  - Higher bus utilization with two masters.
    - DMA transfers have lower latency, the CPU is no longer the middle man.



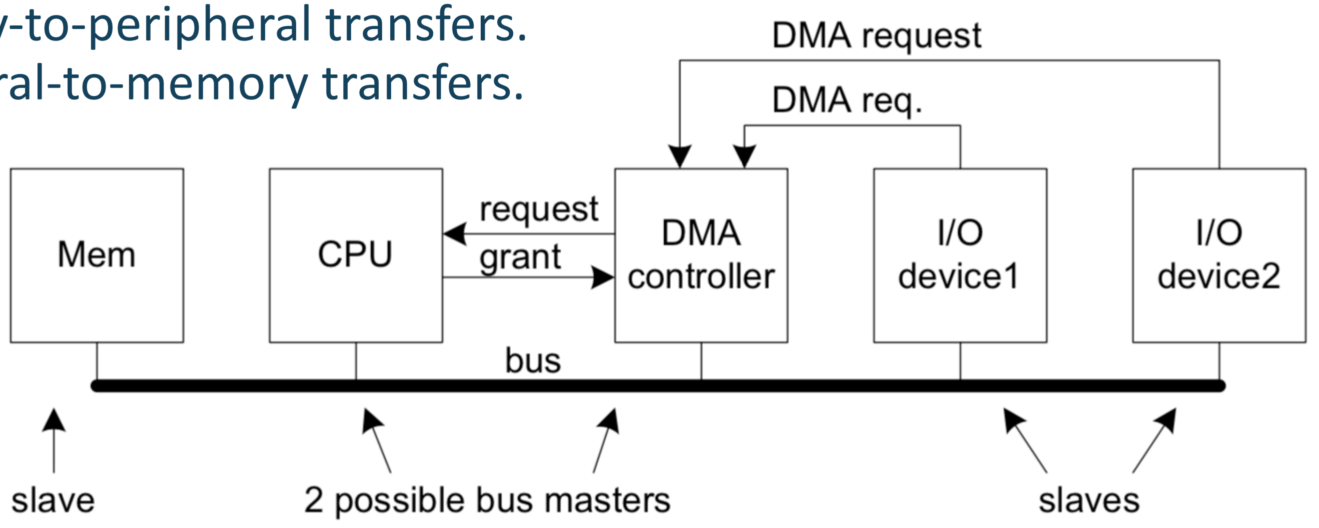
Multi-Master Bus Control: Bus Arbiter

- Some devices may employ a central bus arbiter instead of a DMA controller.
  - It receives requests for all the masters and grants access based on system priorities.
  - Pro: Each peripheral in this case becomes a master instead of a slave.
    - This provides the fastest direct access to the memory.
    - Used for high performance peripherals like USB, Ethernet, PCI-Express...
  - Con: This can increase complexity and is more expensive, as now peripherals need to operate in both master and slave modes.



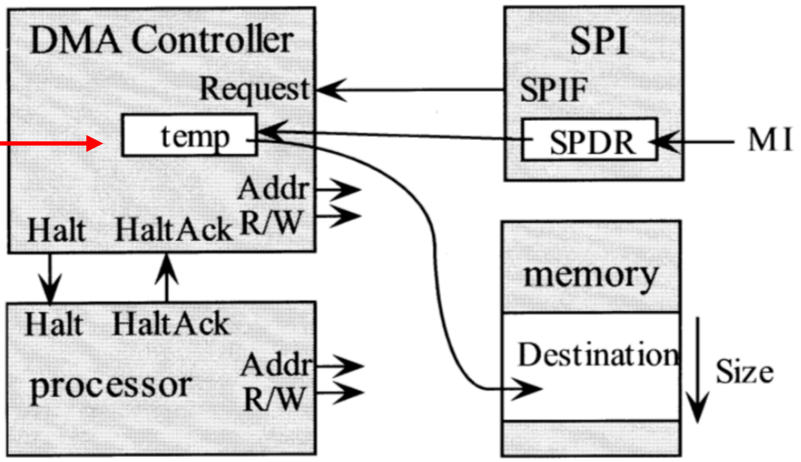
Multi-Master Bus Control: DMA Controller

- On the TM4C1294, uDMA controller is only granted bus access when the bus is otherwise idle.
  - The uDMA controller never holds up accesses by the CPU.
  - Other devices may allow the DMA to halt the CPU affecting performance, it is implementation specific.
- The controller is the bus master during DMAs
  - It determines the priority of all outstanding DMA requests and which slave peripherals will have bus access.
  - It controls the transfer of data between peripherals and memory.
- The TM4C1294 DMA controller can perform
  - Memory-to-memory transfers.
  - Memory-to-peripheral transfers.
  - Peripheral-to-memory transfers.

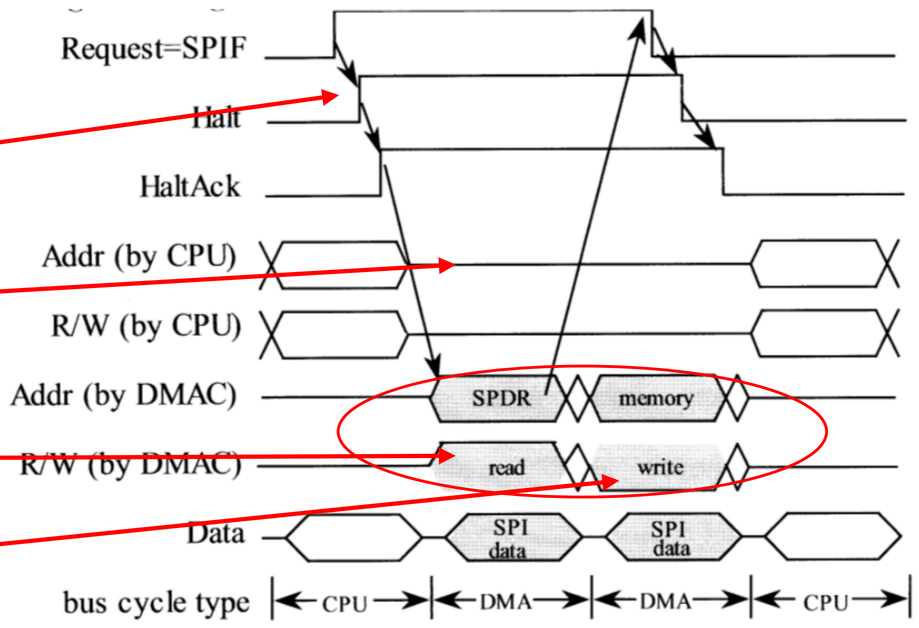


Example DMA Bus Transaction

- There are many ways to implement a DMA controller.
  - Some implementations of DMA controllers will temporally copy data into the controller. (Dual Address)
    - This will provide lower performance than the bus arbiter.
    - Requires two transactions versus one direct transaction.
  - Others DMAs will apply the address and the peripheral will drive the data directly (single address / single transaction)



- One possible implementation: example of a SPI read (dual address)
  - The DMA Controller requests the bus from the CPU.
    - This triggers a Halt request.
  - If the CPU is not using the bus,
    - It acknowledges the halt and releases bus control.
  - The DMA takes control of the bus.
    - First reading from the SPI interface into a temporary register.
    - Second writing to the memory .
  - The CPU will take back control when it needs it or if the DMA is complete.



Common DMA Mode Choices

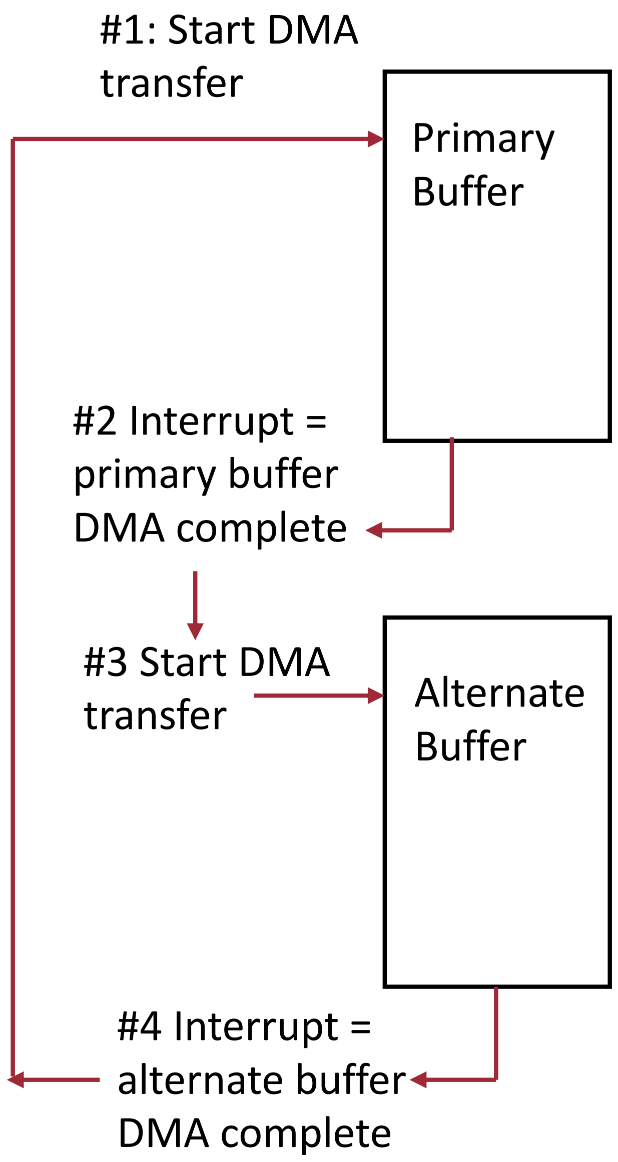
- What initiates the DMA transfer?
  - Software trigger, input or output peripheral, periodic timer.
- Type of transfer:
  - Burst modes will execute multiple consecutive DMA transactions.
  - Cycle steal modes will interleave DMA and CPU transactions.
- Autoinitialization mode:
  - Single event or continuous transfer.
- Precision (word size):
  - 8, 16 or 32-bit words.
- Address mode:
  - Dual or Single address (see previous page)
- Priority
  - How should CPU behave when a while a DMA is on going?
  - Should CPU be unaffected? Should CPUhalt? Should CPU interrupts be serviced?
- Synchronization
  - Should a flag be set or an interrupt asserted when DMA completes?

TM4C1294 uDMA Configuration

- A channel is configured for each type of DMA operation.
- The channel has configurations options.
  - How to start the DMA: Software triggered or interrupt driven.
  - Priority of the channel, higher priorities are executed first.
  - Source address
  - Destination address
  - Word size – 8, 16, 32-bits.
  - Transfer size: 1 – 1024 words in a DMA operation.
  - Should the addresses increment, decrement or stay constant during a transfer of multiple words.
  - Channel transfer modes
    - Basic / Auto transfer: a single request copies from source to destination.
    - Ping-pong : accommodate constant streaming of data to or from a peripheral using two alternating buffers.
    - Scatter-gather (up to 256 arbitrary transfers with a single request)

Ping-Pong Mode

- Ping-pong mode allows for continuous DMA transfers to occur.
- Two buffers are setup, a primary and an alternate buffer.
  - Both buffers are configured to perform a DMA transfer between buffer memory and the peripheral.
- The DMA transfer is started on the primary buffer.
  - When the transfer is complete it stops and an interrupt occurs to signal the primary buffer transfer is done.
- When restarted, the DMA transfer continues using the alternate buffer while the data in the primary buffer is processed.
  - When the alternate buffer transfer is complete it stops and an interrupt occurs to signal the alternate buffer transfer is done.
- When restarted, the DMA transfer returns to the primary buffer while the data in the alternate buffer is processed.
- This allows for continual data transfers without shared data problems.
  - The data is transferred using primary buffered, while the alternate buffer data is being processed.
  - The data is transferred using the alternate buffer, while the primary buffer data is being processed.
  - Transfers to primary and alternate buffers can switch back and forth indefinitely.


#1: Start DMA transfer
Primary Buffer
#2 Interrupt = primary buffer DMA complete
#3 Start DMA transfer
Alternate Buffer
#4 Interrupt = alternate buffer DMA complete