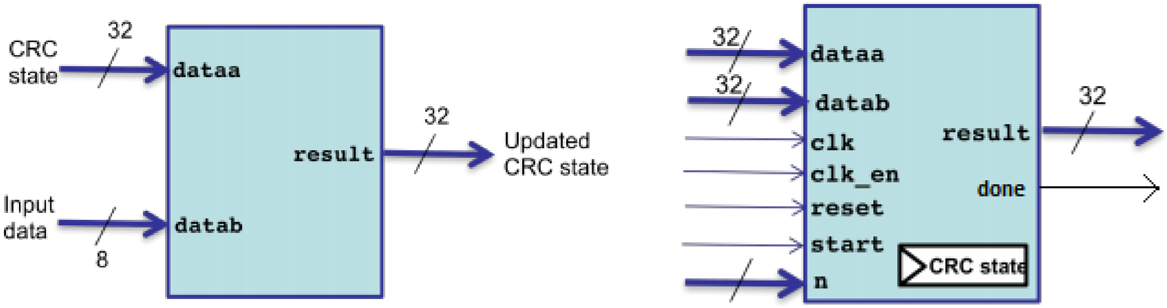
CRC CUSTOM INSTRCUTION DESIGN

BLOCK DIAGRAM



(a) Combinational Custom Instruction HW

(b) Extended multi-cycle Custom Instruction HW

COMBINATIONAL DESIGN (CI COMBINATIONAL)

In combinational design, the combinational (single cycle) custom instruction implements one step of CRC computation. The instruction takes as inputs the 32-bit CRC and 8-bits of data, and produces the updated CRC. The final XOR operation ($crc\_temp = crc\_temp$ ^ $0xFFFFFFFF$) is implemented in software by flipping all the bits of *crc_temp*. The CRC table is stored internally to the custom instruction HW.

MULTICYCLE DESIGN (CI MULTICYCLE)

In multicycle design, the extended multicycle custom instruction supports 6 operations – loading the value of CRC into custom instruction HW, computing CRC on 1-byte of data, computing CRC on 2-bytes of data, computing CRC on 3-bytes of data, computing CRC on 1-word of data, reading the value of CRC from the custom instruction HW.

The wait cycles for 2-bytes CRC computation, 3-bytes CRC computation, 1-word CRC computation are 2 cycles, 3 cycles, 4 cycles respectively. Otherwise, it is a 1-cycle wait (basically, processor can process new instruction the following cycle). In this way, the extended multicycle design implements variable latency.

Software loads the initial CRC value into the custom instruction HW. If the start or end bytes of payload/frame are unaligned, they are handled by performing sub-word CRC computations (on 1-byte or 2-byte or 3-byte of data) in the custom instruction HW. For the remaining bytes of payload/frame, the software performs CRC computation on 1-word of data in the custom instruction HW. The updated CRC is read out from the custom instruction HW and is flipped to get the final CRC value.

EXTRA CREDIT DESIGN (CI ADVANCED)

In extra credit design, the extended multicycle custom instruction is enhanced to support 7 operations – the 6 operations of the multicycle design and additionally, computing CRC on 2-words of data.

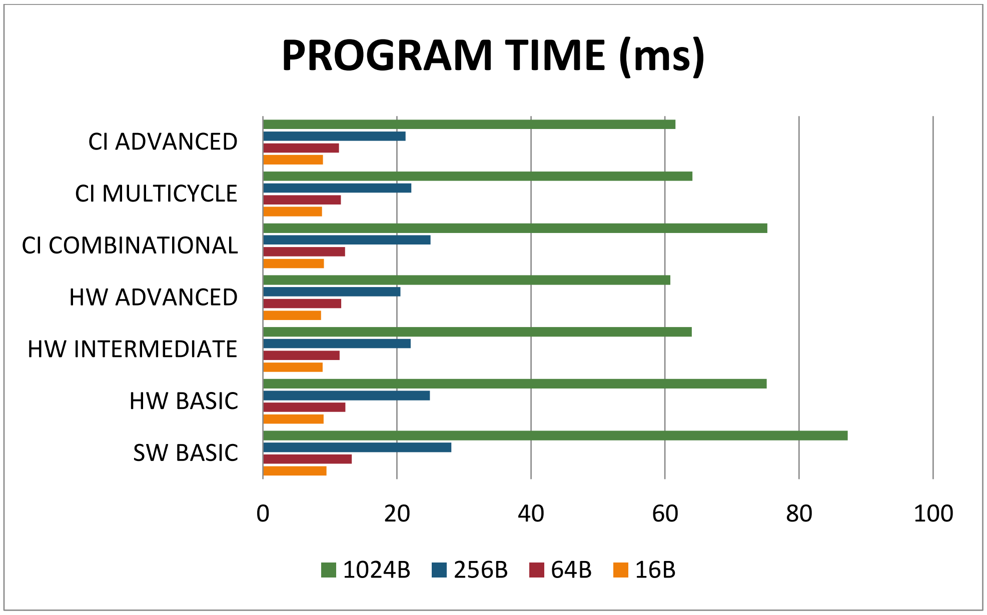
The wait cycles for the 2-words CRC computation is 8 cycles. The other wait cycles are same as in the multicycle design.

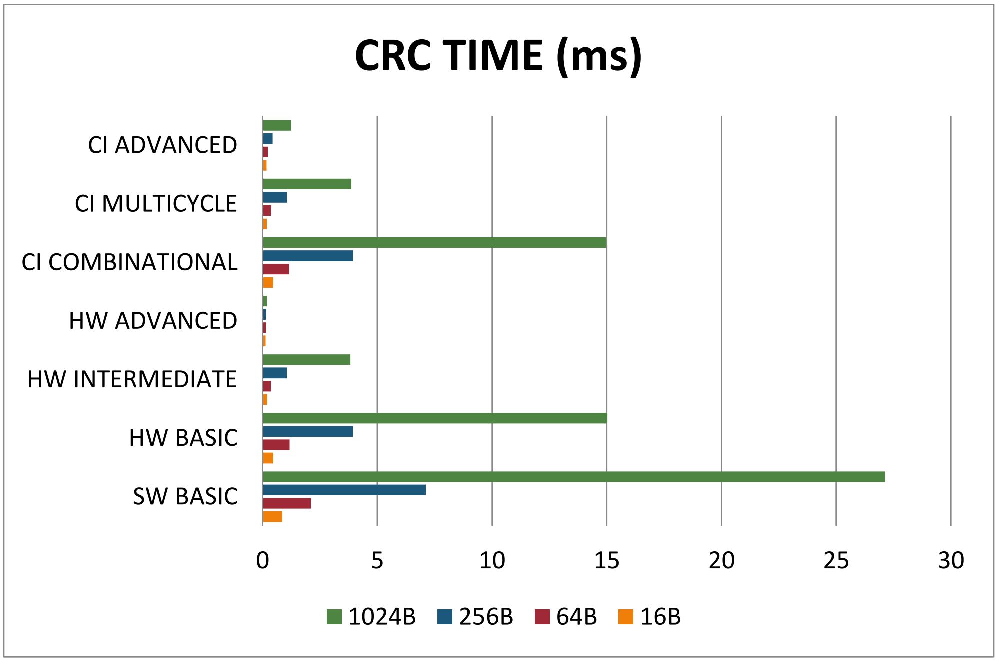
Software is also enhanced to support sending doubleword data to the custom instruction HW when possible. In addition, software uses loop unrolling technique to enhance the performance. For larger payloads/frames, it sends blocks (in the multiples of 16, 8, 4, 2 or 1) of doubleword data. If the remaining bytes in the payload/frame cannot be sent in blocks of doubleword data, it sends word by word to the custom instruction HW. The start and end bytes of payload/frame, if unaligned, are handled in the similar manner as in the multicycle design.

RESULTS

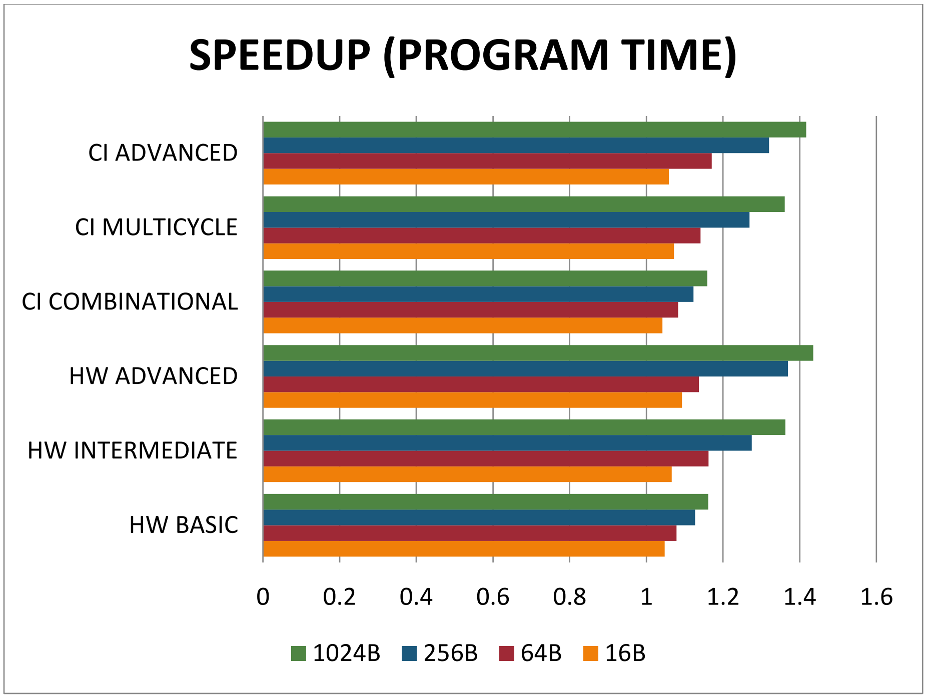
The execution time of the program, CRC computation (ICV time + FCS time) is measured for a single frame of different payload lengths – 16B, 64B, 256B and 1024B. **Note that if the payload length is chosen as a multiple of 4, the frame length used for FCS computation is not a multiple of 4.**

The execution times shown below are measured across the reference software implementation (**SW BASIC**), primitive CRC accelerator (**HW BASIC**), Design IDEA1 (**HW INTERMEDIATE**), Design IDEA2 (**HW ADVANCED**), combinational design (**CI COMBINATIONAL**), multicycle design (**CI MULTICYCLE**) and extra credit design (**CI ADVANCED**) and speedup achieved with respect to the reference software implementation is shown for both – the total program time and the CRC computation time.

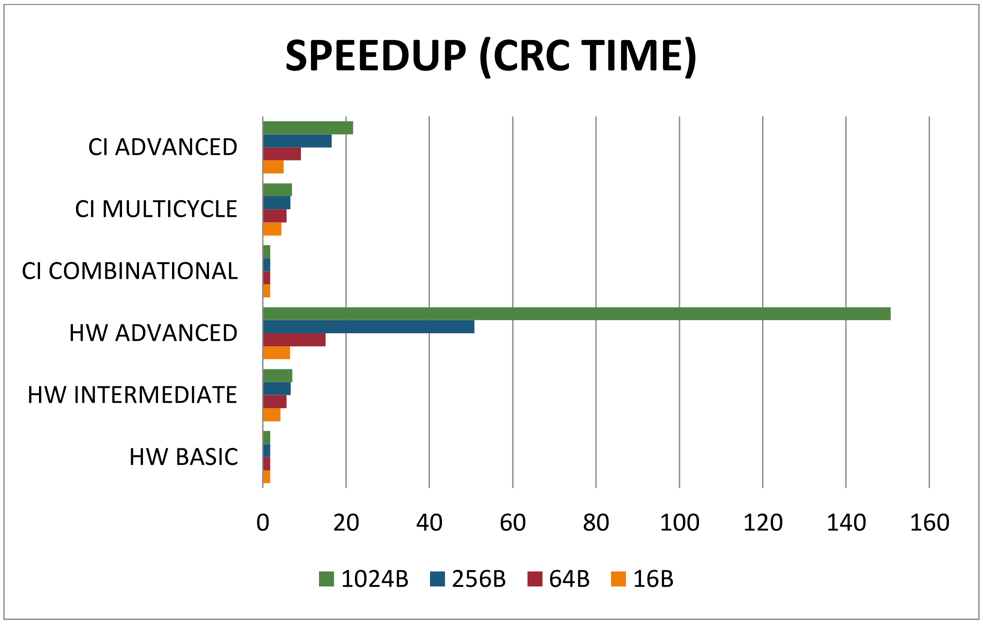




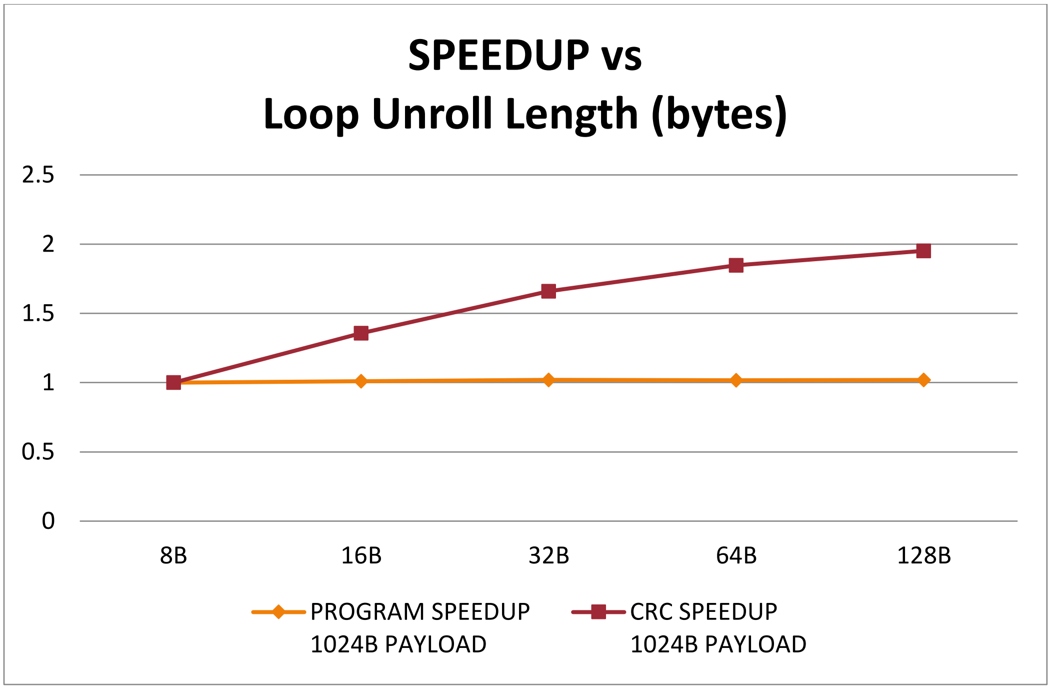
Further, the SWBasic, HWBasic, HWIntermediate, HWAdvanced, CICombinational, CIMulticycle and CIAdvanced files under the profiling/ directory give the performance counter report for all the above scenarios. To rerun the above scenarios, set the mode variable in software/demo/misc.h to 0x1 for SW BASIC, 0x2 for HW BASIC, 0x3 for HW INTERMEDIATE, 0x4 for HW Advanced, 0x5 for CICombinational, 0x6 for CIMulticycle and 0x7 for CIAdvanced.



It is noticed that the overall program benefits only by **5%-45%** from the Hardware Accelerator/Custom Instruction for CRC and their different design optimizations.



The combinational custom instruction design speeds up the CRC evaluation by **1.8x**. Whereas, the extended multicycle custom instruction design achieves a speedup of almost **4x**. Of the 3 strategies for the custom instruction design, the extra credit custom instruction design strategy outperforms the extended multicycle custom instruction design by about **2x-3x**.



In addition, the speedup of program and CRC is studied for 1024B payload by varying the extent of loop unrolling length (bytes). The baseline comparison is made with respect to the single doubleword CRC computation time for the 1024B payload. It is observed that the overall program time is not affected at all by the loop unrolling. However, the CRC computation time improves almost by **2x** when the loop is unrolled **16** times to transfer 16 doublewords (**128 Bytes**). It increases almost linearly with the increase in the loop unrolling length.

The combinational custom instruction achieves the same performance (**1.8x**) as the primitive design for the HW accelerator. *The combinational custom instruction does not stall the processor pipeline and so is the HW accelerator. The processor and the HW accelerator can operate concurrently. The processor sends the payload/frame data to the HW accelerator through the Avalon bus. However, this does not halt the processor from processing subsequent instructions.*

The extended multicycle custom instruction also achieves similar performance (**7x**) as the Design IDEA1 for the HW Accelerator. *The extended multicycle custom instruction stalls the processor pipeline (for 3 more cycles) for the CRC computation on each word of payload/frame data. These additional cycles are negligible (**< 1%**) when compared to the total cycles spent on the CRC computation (**194K cycles**). In the case of HW accelerator, the processor and HW accelerator can operate* *concurrently. The processor sends data word by word into the synchronous FIFO inside the HW accelerator and hence, avoids stalling of processor pipeline.*

Unlike the other strategies, the Design IDEA2 for the HW accelerator achieves tremendous speedup (**~150x vs ~21x**) in comparison to the extra credit custom instruction for larger payloads/frames. *Loop Unrolling and Doubleword CRC computations in extra credit design improve the performance by decreasing the number of branching. However, the Design IDEA2 for the HW accelerator employs DMA to transfer the payload/frame data from the memory to the HW accelerator. DMA does this data transfer in the most efficient manner.*

In summary, the HW accelerator approach outperforms the custom instructions approach by significant margin with better design optimizations. In general, the custom instructions stall the processor pipeline. But, the HW accelerator can operate concurrently with the processor. Also, the custom instruction approach limits the number of inputs and outputs that a custom instruction HW can handle at a time. For large scale designs with multiple inputs and outputs, HW accelerator is the better approach to HW/SW partitioning.

APPENDIX

PRIMITVE DESIGN (HW BASIC)

In the primitive design, the CRC module interfaces with the Avalon bus through the standard Avalon Slave Interface.

The processor sends the payload/frame data for CRC computation one byte at a time. A single byte of payload/frame data is sent in the lower byte of the 32-bit *writedata* bus by writing to the CRC_CONTROL register (CRC_AVALONSS_INTERFACE_0_BASE+8). In addition, the bit 31 must be set by software to enable the CRC computation on the single valid data byte (CRC_CONTROL [7:0]).

The CRC is computed on the following clock cycle using the CRC table stored in the internal memory of the CRC module. The current value of the CRC can be read at any time by reading the CRC_VALUE register (CRC_AVALONSS_INTERFACE_0_BASE). The final value of CRC is read from the CRC_STATUS register (CRC_AVALONSS_INTERFACE_0_BASE+4). The CRC value is reset by writing 0x1 to CRC_STATUS register every time a CRC computation needs to be performed on a new set of payload/frame.

DESIGN IDEA1 (HW INTERMEDIATE)

In Design IDEA1, a 32-deep synchronous FIFO sits between the Avalon Slave interface and the CRC module.

The processor now sends the payload/frame data for the CRC computation 4 bytes at a time. A word of payload/frame data is sent in the *writedata* bus by writing to CRC_AVALONSS_INTERFACE_0_BASE. This data is stored in the synchronous FIFO. When full, the transaction is stalled by asserting the *waitrequest* until the FIFO has the capacity to accept the incoming transaction.

When the FIFO is not empty, the CRC module reads from the FIFO and takes 4 cycles to compute the CRC on the data word. Hence, FIFO is read every 4 cycles by the CRC module.

Another key design strategy is to handle different payload/frame lengths. The software writes to the CRC_AVALONSS_INTERFACE_0_BASE one word at a time if there is a valid payload/frame data available. These writes fill the synchronous FIFO. CRC module reads from this FIFO every 4 cycles to compute CRC on a single word of data.

For the last few bytes (< 4) of the payload/fame data, the software writes to the CRC_CONTROL register one byte at a time (like the primitive design). The CRC module is upgraded so that the writes to CRC_CONTROL register is stalled when the CRC module has not finished with processing all the contents of the synchronous FIFO.

In addition, the number of memory reads from the processor is reduced by reading one word at a time. This can have a significant impact on the performance of CRC and overall program as the payload and frames are read one byte at a time from memory. Care must be taken if the starting and ending addresses are not word aligned so that invalid data are ignored in the software itself. For these unaligned start and end data bytes (< 4), processor can use the CRC_CONTROL register to do CRC computation one byte at a time.

The above design strategy reduces the number of accesses to/from the memory by approximately 75%, when compared to the primitive design. This is because, the processor now reads and writes word by word instead of byte by byte of payload/frame data. In addition, the synchronous FIFO at the interface can help processor to keep writing the payload/frame data even when CRC module is busy internally.

DESIGN IDEA2 (HW ADVANCED)

In Design IDEA2, the strategy is for the processor to read starting few unaligned bytes of payload/frame data from memory and writing byte by byte into CRC_CONTROL register. Then, configure DMA controller to do the data transfer for the payload/frame section in the memory which are word aligned. DMA writes to a fixed location CRC_AVALONSS_INTERFACE_0_BASE to fill the synchronous FIFO. The writes from DMA are efficient and stress the 32-deep synchronous FIFO at the CRC slave interface. The processor can then handle the ending few bytes of frame/payload data that are not word aligned. These bytes are written one by one into CRC_CONTROL register to evaluate the CRC.