FSL - SW Acceleration with MicroBlaze for Multi-Channel Data Acquisition

Xilinx, Inc.
Agenda

• Data Pipe Challenge
• Introduction to Fast Simplex Link (FSL)
• FSL Description
  – Hardware Architecture
  – Software Instructions
  – FSL Design Tips
• Platform Studio Design Flow for FSL
• Design Example, IDCT processor
• Summary and Conclusion
• Reference
The Software Streaming Data Challenge!

- We need to move data through a hardware/software processing application
- Data may be of a streaming or burst nature
- Greater MIPS demands on processors
- New software algorithms combine RISC and DSP flavors
- Deterministic latency between hardware and software
Solution Alternatives

• Bus peripheral, possibility OPB
  – Multiple clock cycle overhead
  – Address decode time
  – Arbitration, loss of hardware/software coherency

• Custom microprocessor instruction access to peripheral hardware
  – May require processor to be stalled
  – Complex logic can slow overall processor speed
  – May require assembly language to access special instruction that is not pre-defined in standard “C” compiler
The Xilinx Alternative

- Platform FPGA with soft core processor (SCP)
- Flexibility in hardware design, re-programmable
- MicroBlaze Soft Core Processor
  - Fast Simplex Link (FSL) interface, application dedicated as opposed to bus structured peripheral
  - Point-to-point link into processor core pipeline, no transaction overhead
  - Alternative to custom microprocessor instruction set
  - Pre-defined “C” routines, simple and easy to use
Agenda

• Data Pipe Challenge
• Introduction to Fast Simplex Link (FSL)
• FSL Description
  – Hardware Architecture
  – Software Instructions
  – FSL Design Tips
• Platform Studio Design Flow for FSL
• Design Example, IDCT Processor
• Summary and Conclusion
• Reference
MicroBlaze FSL

- Uni-directional point-to-point FIFO-based communication
- Dedicated (unshared) and non-arbitrated architecture
- Dedicated MicroBlaze C and ASM instructions for easy access
- High speed access in as little as two clocks on processor side, 600MHz at hardware interface
- Available in Xilinx Platform Studio (XPS) as a bus interface library core
MicroBlaze Software Acceleration and FSL

Performance Improvement

- MP3 Decoding with Custom Hardware Logic
  - Streaming co-processing using FSL to connect MicroBlaze to hardware accelerator
  - MP3 decoding improves dramatically
  - FSL stream-based DMA access to memory and hardware accelerator

<table>
<thead>
<tr>
<th>Custom Hardware Logic</th>
<th>100MHz MicroBlaze, pure software</th>
<th>100MHz MicroBlaze + FSL + IMDCT</th>
<th>100MHz MicroBlaze + FSL + DCT</th>
<th>100MHz MicroBlaze + FSL + LL MAC</th>
<th>100MHz MicroBlaze + FSL + DCT + IMDCT + LL MAC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>= 290 seconds</td>
<td>= 190 seconds</td>
<td>= 170 seconds</td>
<td>= 35 seconds</td>
<td>= 7 seconds</td>
</tr>
<tr>
<td></td>
<td>1X</td>
<td>1.5X</td>
<td>1.7X</td>
<td>8.2X</td>
<td>41X</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FSL Architecture

- FSL core illustrating simplified inputs and outputs
- Internal FIFO, control bit and data path configured using the FSL Wizard
FSL Highlights

- FSL architecture is automatically generated based on user requirements. Required resources from 21 to 451 slices
- Core can be configured as master or slave
- Separate control bit channel
- MicroBlaze allows for up to eight parallel FSL channels
- Easy to use “C” calls for MicroBlaze FSL channels
  - Pre-define compiler functions
  - Optional use of assembler language
- Simple and fast interface
FSL Features

• Configurable data bus widths – 8, 16, 32 bits
• Configurable FIFO depths – 1 to 8193 using SRL16 or block RAM
• Synchronous or asynchronous FIFO clocking with respect to the MicroBlaze system clock
• Selectable use of control bit
• Blocking and non-blocking software instructions for data and control (get and put)
• Simple software interface using pre-defined “C” instructions; Automatically generated C drivers.
Why FSL?

• It’s simple, fast, and easy to use.
• FSL vs. Custom uP instruction.
  – Custom hardware not dependent on instruction decode
  – Clock speed not slowed down by new hardware
  – Minimal FPGA fabric overhead
  – No need to stall the processor because of custom function clock latency
  – No need to modify the C compiler (Could be risky!)
FSL Advantages

• No need to learn new bus architectures to build a hardware interface
• Saves clock cycles – faster than a bus interface
  – Eliminates bus signaling overhead.
    • No arbitration
    • No address decode
    • No acknowledge cycles
  – Decoupled data clock from CPU allows for asynchronous operation
• Control bits limit need for a complex interrupt structure
• FSL port standard promotes design reuse
When to Use a FSL

- Direct interface to custom hardware data and control lines
- Requirement for high-speed pipe into processor when shared bus interface is inadequate
- Alternative to using a DMA structure
  - Flexibility to compute incoming data stream rather than DMA-imposed block processing
  - Eliminate bus competition
- Software applications that favor stream processing
FSL Applications

• Communications – data and frame processing
• Audio/Video compression/decompression engines (See MP3 live audio demo on Spartan-3 board)
• Industrial Control
  – Short control loop time
  – Fast output response to inputs
  – Motion Estimation
FSL Applications

• Data Acquisition
  – Burst and continuous stream
  – Multiple channels and data/control input types

• High-speed communications link between processors; FSL to FSL links two MicroBlaze processors
  – FSL to local memory, with semaphore control
Agenda

• Data Pipe Challenge
• Introduction to Fast Simplex Link (FSL)
• **FSL Description**
  – Hardware Architecture
  – Software Instructions
  – FSL Design Tips
• Platform Studio Design Flow for FSL
• Design Example, IDCT Processor
• Summary and Conclusion
• Reference
FSL: Synchronous FIFO, Implemented with SRL16

LUT configured as 16bit Shift Register LUT (SRL16)

FSL_M_Data [0:31]
FSL_M_CLK
FSL_S_Read
Read Address Counter
16 deep
FSL_M_Full
FSL_M_Control
FSL_S_Control
FSL_S_Data [0:31]
FSL_S_Exists
FSL_S_Read
FIFO
32-bit data
FIFO Length
Status
Control
Address Counter and FIFO Count Compares
FSL: Synchronous FIFO, Implemented with Dual Port BRAM

Write Address Counter

Read Address Counter

Write Port Logic

Read Port Logic

FSL_S_Data [0:31]

FSL_M_Data [0:31]

FSL_M_CLK

FSL_M_Write

FSL_M_Control

FSL_S_Control

FSL_M_Full

FSL_M_Read

FSL_S_Full

FSL_S_Read

FSL_S_Exists

FSL_S_Control

Address Counter and FIFO Count Compares

FIFO Length

Status

Control
FSL Interface

• Simple, fast learning curve
• One “symmetric” core for master or slave, MicroBlaze hooks to one side, your application hooks to the other
• Six parameters, set in XPS
  – C_FSL_DWIDTH - Data width – 8, 16, 32 bits
  – C_FSL_DEPTH - FIFO depth, 1 to 8193 words
  – C_USE_CONTROL - Control bit use enable
  – C_ASYNC_CLKS - FIFO clocking, sync or async
  – C_IMPL_STYLE - FIFO SRL16 or block ram
  – C_EXT_RESET_HIGH - External reset polarity
FSL Interface

• Core signals
  – FSL_Clk – Master/slave clock used only in synchronous mode
  – SYS_Rst – Core reset input
  – FSL_Rst – Core reset output to user application

• Master side signals
  – FSL_M_Clk – Master clock used only in asynchronous mode
  – FSL_M_Data – Input data bus
  – FSL_M_Control – Input control bit, when enabled
  – FSL_M_Write – FIFO write (push) enable
  – FSL_M_Full – FIFO full flag
FSL Interface

• Slave side signals
  – FSL_S_Clk – Slave clock used only in asynchronous mode
  – FSL_S_Data – Output data bus
  – FSL_S_Control – Output control bit, when enabled
  – FSL_S_Read – FIFO read (pop) enable
  – FSL_S_Exists – FIFO “not empty” flag

• When C_ASYNC_CLKS is ‘0’, synchronous mode, use FSL_Clk

• When C_ASYNC_CLKS is ‘1’, asynchronous mode, use FSL_M_Clk and FSL_S_Clk
Agenda

• Data Pipe Challenge
• Introduction to Fast Simplex Link (FSL)
• FSL Description
  – Hardware Architecture
  – Software Instructions
  – FSL Design Tips
• Platform Studio Design Flow for FSL
• Design Example, IDCT Processor
• Summary and Conclusion
• Reference
FSL Control Software

• Eight dedicated MicroBlaze instructions (assembler)
  – get, put, nget, nput
  – cget, cput ncget, ncput

• Block vs. non-blocking, an important FSL feature
  – **Blocking** stalls the MicroBlaze from completing the above instruction until data is available for reading or writing. This can happen when the FIFO is empty or full during a read or write, respectively
  – **Non-blocking** completes the MicroBlaze instruction in two clock cycles, regardless of the state of the FIFO flags.
FSL “C” Interface

• Assembly is not a language of choice
• All eight MicroBlaze are implemented as pre-defined macros
  – microblaze_bread_datafsl(val, id)
  – microblaze_bwrite_datafsl(val, id)
  – microblaze_nbread_datafsl(val, id)
  – microblaze_nbwrite_datafsl(val, id)
  – microblaze_bread_cntlfsl(val, id)
  – microblaze_bwrite_cntlfsl(val, id)
  – microblaze_nbread_cntlfsl(val, id)
  – microblaze_nbwrite_cntlfsl(val, id)
Agenda

• Data Pipe Challenge
• Introduction to Fast Simplex Link (FSL)
• **FSL Description**
  – Hardware Architecture
  – Software Instructions
  – **FSL Design Tips**
• Platform Studio Design Flow for FSL
• Design Example, IDCT Processor
• Summary and Conclusion
• Reference
FSL Interface Design Techniques

• Data path should be serial, as opposed to block, of word width to facilitate FIFO usage

• Generous use of pipeline registers to maintain a high clock rate – clock period vs. system latency

• Use multiple FSL channels and control bits
  – FSL data channel
  – FSL control channel, application command & status
  – Control bit vs. interrupt use, i.e. new data, BOT, EOT
  – Interrupts typically have a larger software latency

• Consider design reuse for next project
FSL Interface Design Considerations

• Application type – software algorithm
  – Streaming data, continuous flow through algorithm
  – Block data, application of algorithm on data set

• Timing – when do things happen?
  – Controlled by system timer
  – Periodic external input
  – Data-dependent, expected pattern or result

• Data rate – streaming or burst
FSL Coding Techniques

• Watch your “C” to insure speed, and memory use efficiency
  – Review assembly code output from compiler, system.a
  – Use registered variables and/or data side cache
  – Ultimate control, write code in assembler, Ugg!
    ```
    asm("get %0, %1" : "=d" (##val##) : "m" (rfsl1))
    asm("put %1, %0" : "=m" (rfsl1) : "d" (##val##))
    ```

• Assembler coding tips
  – Non-blocking operations set the carry bit if data transfer fails, use branch instructions that look at the carry for loop control
  – Control bit failures set the FSL_error bit

• Avoid subroutine calls and superfluous instructions in FSL operation loops that will take more instructions to execute
System Considerations When Using FSL

• Deterministic hardware/software latency
  – Control application hardware data rate and software loop timing so that FIFO does not empty or fill
    • A FIFO that empties indicates that the data rate could be increased
    • A FIFO that goes full indicates data overrun and loss, tighten up software loop code or adjust data rate
  – Select FIFO size base on burst nature of data rate, just large enough to insure FIFO will overflow
  – Synchronization between hardware and software
  – Data pipeline, starting up, and ending. (fill and flush)
System Considerations When Using FSL

- Limit Interrupt usage, high overhead for context switching, use FSL control bit instead
- Keep software loop access to FSL tight, avoid subroutine or device driver calls to FSL instructions
- As most systems require asynchronous clocking, keep MicroBlaze system clock domain separate from backend hardware clock domain; FSL is used as the only instrument to cross clock boundaries
- Consider system latency in both hardware and software when evaluating real-time performance
Agenda

• Data Pipe Challenge
• Introduction to Fast Simplex Link (FSL)
• FSL Description
  – Hardware Architecture
  – Software Instructions
  – FSL Design Tips
• **Platform Studio Design Flow for FSL**
• Design Example, IDCT processor
• Summary and Conclusion
• Reference
FSL Design Flow

FSL Wizard

- XPS - Engage FSL Design Wizard
- Present FSL channels/cores are displayed
- Use Add New to instantiate a new FSL core
FSL Design Wizard Flow

• Give the core a name
• Setup sample software routine function parameters
  – Inputs and outputs are 32 bit integer arrays of argument size
  – Inputs are put into FSL FIFO
  – Outputs are get from FSL FIFO
  – Sample “C” program in drivers\core_name\src directory
FSL Design Wizard Flow

- Core summary, generate with Finish
- New core automatically associated with next FSL channel
- Core is instantiated into design upon OK
FSL Design Wizard Flow

• Sample “C” code generated as a foundation of FSL application driver

• Hardware HDL code generated as skeleton for backend FSL application and includes:
  – VHDL code
  – MPD hardware description
  – PAO analyze order
FSL Design Wizard Flow

New FSL peripheral is automatically instantiated into your design.
FSL Design Flow

Custom Hardware Peripheral

• Good synchronous design practice
• Design FSL interface as described earlier
  – Determine number of FSL channels needed for application
  – Master writes (push) to FIFO
  – Slave reads (pop) from FIFO
• FSL wizard auto generates design template in XPS *pcores* directory
  – Generates MPD/PAO files describing your FSL peripheral
  – Generates and connects FSL signal names to top level of VHDL entity or Verilog module
  – Adds the new peripheral to your project
  – You add in custom hardware and bring out I/O ports
FSL Design Flow

Custom Peripherals Directory

• Peripheral list is created from pcores directory
• Data directory contains MPD and PAO files
• hdl directory contains VHDL or Verilog sub-directory with hdl code files, hierarchy is determined in PAO file.
• Be sure to follow signal naming conventions
BEGIN fsl_fpu, IPTYPE=PERIPHERAL, HDL=VHDL, EDIF=TRUE

# Define bus interface
BUS_INTERFACE BUS=SFSL, BUS_STD=FSL, BUS_TYPE=SLAVE
BUS_INTERFACE BUS=MFSL, BUS_STD=FSL, BUS_TYPE=MASTER

# Global ports
PORT Clk = "", DIR=IN, SIGIS=CLK
PORT Reset = "", DIR=IN

# FSL signals to Custom Hardware design
PORT FSL_S_CLK = FSL_S_Clk, DIR=out, SIGIS=CLOCK, BUS=SFSL
PORT FSL_S_READ = FSL_S_Read, DIR=out, BUS=SFSL
PORT FSL_S_DATA = FSL_S_Data, DIR=in, VEC=[0:31], BUS=SFSL
PORT FSL_S_CONTROL = FSL_S_Control, DIR=in, BUS=SFSL
PORT FSL_S.Exists = FSL_S_Exists, DIR=in, BUS=SFSL

PORT FSL_M_CLK = FSL_M_Clk, DIR=out, SIGIS=CLOCK, BUS=MFSL
PORT FSL_M_WRITE = FSL_M_Write, DIR=out, BUS=MFSL
PORT FSL_M_DATA = FSL_M_Data, DIR=out, VEC=[0:31], BUS=MFSL
PORT FSL_M_CONTROL = FSL_M_Control, DIR=out, BUS=MFSL
PORT FSL_M_FULL = FSL_M_Full, DIR=in, BUS=MFSL

# Custom hardware I/O
Port MYINPUT = "", DIR=IN, VEC=[0:31]
Port MYOUTPUT = "", DIR=OUT, VEC=[0:31]

END
PAO File
Peripheral Analyze Order

• PAO file contains all hdl filenames, in order of hierarchical dependency, that make up your design
• Start with the auto-generated design PAO file to use as template

Three files make up this user peripheral

Library directory in pcores

hdle entity or module file name

```
lib xil_idct_v1_00_a idct
lib xil_idct_v1_00_a idct_core
lib xil_idct_v1_00_a xil_idct_v1_00_a
```
FSL Design Wizard Flow

Summary

• Quick and easy generation of FSL based peripheral
  – Sample “C” FSL driver code provided
  – VHDL skeleton of backend user interface
  – Initial core MPD and PAO files
  – FSL peripheral automatically instantiated in project

• FSL wizard limitations (Today!)
  – Always uses two 32 bit FSL channels, one in, one out
  – May require additional parameters set in Add/Edit Cores
  – Refer to presentation reference section describing the Add/Edit Cores design flow that provides total FSL design flexibility
Agenda

• Data Pipe Challenge
• Introduction to Fast Simplex Link (FSL)
• FSL Description
  – Hardware Architecture
  – Software Instructions
  – FSL Design Tips
• Platform Studio Design Flow for FSL
• **Design Example, IDCT Processor**
• Summary and Conclusion
• Reference
FSL Reference Design

- C only vs. FSL assisted C code comparison using a timer to count clock cycles
- Input is an 8X8 data array
- Output is an IDCT transformed data array
- This example demonstrates the simple nature of using a FSL in a design
Discrete Cosine Transform

- Transforms a signal or image from the spatial domain to the frequency domain
- The spatial domain is in a matrix format
- The transform characteristics are governed by the matrix size and transform coefficients
- DCT is similar to the discrete Fourier transform which transforms from the time domain to the frequency domain
- Less hardware to implement than FFT
- Ideal application for shared software/hardware implementation, therefore, candidate for FSL
FSL Reference Design
Hardware Implementation
FSL Reference Design

• Mathematical requirements for IDCT per element
  – 8 Multiply’s (64 per sample)
  – 7 Add’s (56 per sample)

• Performance results
  – Software only solution; 8928 clocks
  – FSL accelerated solution; < 900 clocks
  – 10X improvement
Spartan-3 Starter Kit for MicroBlaze

$99 US

- Starter Kit Contents
  - Prototyping board
  - Download cable
  - Xilinx WebPACK CD
  - Documentation
  - Reference Designs
Spartan-3 Starter Kit for MicroBlaze

- Evaluation board for fast and easy MicroBlaze prototyping
  - XC3S200-4FT256C Spartan 3; 200,000 gates
  - XCF02S 2Mb Platform Flash configuration PROM
  - 1MB Fast Asynchronous SRAM
  - 4 digit seven segment display
  - PS2 Keyboard/Mouse port
  - 8 slide switches, 4 push buttons, 8 LEDs
  - 15 pin, 3 bit, 8 color VGA Serial Port
  - Three 40 pin Expansion Connectors
  - 50MHz clock oscillator
Agenda

• Data Pipe Challenge
• Introduction to Fast Simplex Link (FSL)
• FSL Description
  – Hardware Architecture
  – Software Instructions
  – FSL Design Tips
• Platform Studio Design Flow for FSL
• Design Example, IDCT Processor
• Summary and Conclusion
• Reference
FSL Benefits

• Choose optimal balance between hardware and software
  – More bytes/clock than pure software solution
  – Perhaps not as fast as a pure hardware solution
  – Optimized for FPGA fabric utilization while maintaining desired application performance

• Quick and easy application design
  – Simple backend interface
  – Easy software coding
  – Design wizard for ease of XPS tool use
Architecture Solutions

• FSL Solution Advantages
  – High-speed data pipe directly into processor
  – Special “C” instructions allow fast program loop operation

• Alternatives to FSL
  – Traditional bus structure
  – Custom microprocessor
Other Solutions

• When not to use FSL, a bus may be a better choice
  – Systems with a single master bus - fast since there is no arbitration
  – Bus peripheral already exists
  – Heavy use of control signals, does not fit a FIFO structure
  – Random access applications
  – Block- or memory buffer-based applications
  – Software algorithms that are not loop orientated
Summary

• FSL is a direct peripheral connection to the MicroBlaze processor
• Up to eight FSL channels can be instantiated
• Each channel is uni-directional; selectable as master/slave
• FIFO-based, 1 to 8193 depth; 8, 16, 32 bits wide
• Synchronous or asynchronous operation selectable
• Fast, data transfer in as little as 2 clocks
• Easy to instantiate using Platform Studio
• Supported in Spartan-II, Spartan-3, Virtex, Virtex-II, Virtex-II Pro, and Virtex-4
FSL Information Sources

• Xilinx XPS Tools
  – EDK Documentation
  – System Generator
    FSL is a supported feature!

• Application Notes
  – XAPP529
    Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel
  – XAPP433
    Web Server Design Using MicroBlaze Soft Processor
  – FSL-implemented PS2 Keyboard
Agenda

• Data Pipe Challenge
• Introduction to Fast Simplex Link (FSL)
• FSL Description
  – Hardware Architecture
  – Software Instructions
  – FSL Design Tips
• Platform Studio Design Flow for FSL
• Design Example, IDCT Processor
• Summary and Conclusion
• Reference
Backend FSL Timing
FSL Backend Interface
Write Operation – FSL Master

FSL Back-to-Back Write Operation
FSL Interface
Read Operation – FSL Slave

FSL Back-to-Back Read Operation
“C” and MicroBlaze Assembly Coding of FSL Instructions
FSL Control Software

• Eight dedicated MicroBlaze instructions (assembler)
  – **get, put**: Blocking Read and Blocking Write of data to the FSL. The control signal is set to '0'.
  – **nget, nput**: Non-blocking Read and Non-blocking Write of data to the FSL. The control signal is set to '0'.
  – **cget, cput**: Blocking Read and Blocking Write of data to the FSL. The control signal is set to '1'.
  – **ncget, ncp**ut**: Non-blocking Read and Non-blocking Write of words to the FSL. The control signal is set to '1'.

• Block vs. non-blocking, an important FSL feature
  – **Blocking** stalls the MicroBlaze from completing the above instruction until data is available for reading or writing. This can happen when the FIFO is empty or full during a read or write, respectively
  – Non-blocking completes the MicroBlaze instruction in two clock cycles, regardless of the state of the FIFO flags.
FSL Control Software
“C” Language Interface

• Assembly is not a language of choice
• All eight MicroBlaze are implemented as pre-defined macros
  – microblaze_bread_datafsl(val, id) get data blocking, expect control bit = ‘0’
  – microblaze_bwrite_datafsl(val, id) put data blocking, write control bit = ‘0’
  – microblaze_nbread_datafsl(val, id) get data non-blocking, expect ctrl bit ‘0’
  – microblaze_nbwrite_datafsl(val, id) put data non-blocking, write ctrl bit = ‘0’
  – microblaze_bread_cntlfsl(val, id) get data blocking, expect control bit = ‘1’
  – microblaze_bwrite_cntlfsl(val, id) put data blocking, write control bit = ‘1’
  – microblaze_nbread_cntlfsl(val, id) get data non-blocking, expect ctrl bit ‘1’
  – microblaze_nbwrite_cntlfsl(val, id) put data non-blocking, write ctrl bit = ‘1’

• “val” should be typed as Xuint32 (unsigned long) and holds the value going into or out of the FIFO
• “ID” is the FSL address, 0 to 7, typed as Xuint32
# MicroBlaze Assembly

## Coding of FSL Instructions

### get

- **get**
  - Description: get data from FSL x (blocking)
  - Syntax: `get rD, FSLx`

- **nget**
  - Description: get data from FSL x (non-blocking)
  - Syntax: `nget rD, FSLx`

- **cget**
  - Description: get control from FSL x (blocking)
  - Syntax: `cget rD, FSLx`

- **ncget**
  - Description: get control from FSL x (non-blocking)
  - Syntax: `ncget rD, FSLx`

### put

- **put**
  - Description: put data to FSL x (blocking)
  - Syntax: `put rA, FSLx`

- **nput**
  - Description: put data to FSL x (non-blocking)
  - Syntax: `nput rA, FSLx`

- **cput**
  - Description: put control to FSL x (blocking)
  - Syntax: `cput rA, FSLx`

- **ncput**
  - Description: put control to FSL x (non-blocking)
  - Syntax: `ncput rA, FSLx`
MicroBlaze GET Instruction

Description: MicroBlaze will read from the FSLx interface and place the result in register rD. The get instruction has four variants. The blocking versions (when ‘n’ bit is ‘0’) will stall MicroBlaze until the data from the FSL interface is valid. The non-blocking versions will not stall MicroBlaze and will set carry to ‘0’ if the data was valid and to ‘1’ if the data was invalid. The get and nget instructions expect the control bit from the FSL interface to be ‘0’. If this is not the case, the instruction will set MSR[FSL_Error] to ‘1’. The cget and cget instructions expect the control bit from the FSL interface to be ‘1’. If this is not the case, the instruction will set MSR[FSL_Error] to ‘1’.

Pseudocode:

\[
\begin{align*}
(rD) & \leftarrow \text{FSLx} \\
\text{if } (n = 1) \text{ then} & \\
\quad \text{MSR}[\text{Carry}] & \leftarrow \text{not (FSLx Exists bit)} \\
\quad \text{if } ((\text{FSLx Control bit}) == c) \text{ then} & \\
\quad \quad \text{MSR}[\text{FSL_Error}] & \leftarrow 0 \\
\quad \text{else} & \\
\quad \quad \text{MSR}[\text{FSL_Error}] & \leftarrow 1
\end{align*}
\]

Registers Altered:

- rD
- MSR[FSL_Error]
- MSR[Carry]

Latency: 2 cycles if non-blocking or if data is valid at the FSL interface. For locking instruction, MicroBlaze will stall until the data is valid.
MicroBlaze PUT Instruction

Description: MicroBlaze will write the value from register rA to the FSLx interface. The put instruction has four variants. The blocking versions (when ‘n’ is ‘0’) will stall MicroBlaze until there is space available in the FSL interface. The non-blocking versions will not stall MicroBlaze and will set carry to ‘0’ if space was available and to ‘1’ if no space was available. The put and nput instructions will set the control bit to the FSL interface to ‘0’ and the cput and ncpput instruction will set the control bit to ‘1’.

Pseudo code: 

\[(\text{FSLx}) \leftarrow (\text{rA})\]
if (n = 1) then
\[\text{MSR}[\text{Carry}] \leftarrow (\text{FSLx Full bit})\]
\[\text{(FSLx Control bit)} \leftarrow C\]

Registers Altered: \[\text{MSR}[\text{Carry}]\]

Latency: 2 cycles for non-blocking or if space is available on the FSL interface. For blocking, MicroBlaze stalls until space is available on the FSL interface.
FSL Design Flow
Add/Edit Cores
FSL Design Flow
Add/Edit Cores

- Adding the FSL hardware to your MicroBlaze-based system
- In the peripherals tab, add the custom FSL-based IP
FSL Design Flow

Add/Edit Cores

- Add a FSL bus for each channel
- Columns appear for each channel
- Select master or slave for each FSL as connection on the MicroBlaze Side
FSL Design Flow
Add/Edit Cores

- Hook up FSL reset and clock (used in synchronous mode only) for each FSL channel
FSL Design Flow
Add/Edit Cores

- Customize parameters for each FSL channel
- Add parameters to change from default value
- Set each added parameter to desired value
FSL Design Flow
Add/Edit Cores

- Select MicroBlaze parameters
- Select and Add FSL links and data size (if different from 32)
- Set links value for number of FSL channels used
Discrete Cosine Transform
Why Use Discrete Cosine Transform?

• De-correlate the data to remove redundancy
• Further compress by compacting the “energy” in as few coefficients as possible, and discard the rest
• Exploit human characteristics
  – In video and graphics applications, separate data into vision-sensitive and non-vision-sensitive parts
  – In audio applications, separate data into audio discernable components
  – Other applications: image processing, desk top video editing, digital still cameras, surveillance systems and others
2 Dimension 8x8 Discrete Cosine Transform (DCT)

\[ Y_{un} = \frac{1}{4} C_u C_n \sum_{i=0}^{7} \sum_{j=0}^{7} X_{ij} \cos \left( \frac{(2i+1) \pi u}{16} \right) \cos \left( \frac{(2j+1) \pi n}{16} \right) \]

- DCT separates image into parts (spectral sub-bands)
- Like Fourier Transform, DCT transforms signal or image from spatial to frequency domain

2 Dimension 8x8 Inverse Discrete Cosine Transform (IDCT)

\[ X_{ij} = \frac{1}{4} \sum_{u=0}^{7} \sum_{n=0}^{7} C_u C_n Y_{un} \cos \left( \frac{(2i+1) \pi u}{16} \right) \cos \left( \frac{(2j+1) \pi n}{16} \right) \]

- Like inverse Fourier Transform, IDCT transforms signal or image from frequency to spatial domain

Where:
- \( X_{ij} \) is input 8x8 data and \( Y_{un} \) is 8x8 DCT coefficient data
- \( C_u = C_n = 1/2^{1/2} \) for \( u, n = 0 \), otherwise \( C_u = C_n = 1 \)
Matrix for 1D 8-point DCT

\[
\begin{align*}
\hat{Y}_0 &= \hat{C}_{00} C_{01} C_{02} C_{03} C_{04} C_{05} C_{06} C_{07} \\
\hat{Y}_1 &= \hat{C}_{10} C_{11} C_{12} C_{13} C_{14} C_{15} C_{16} C_{17} \\
\hat{Y}_2 &= \hat{C}_{20} C_{21} C_{22} C_{23} C_{24} C_{25} C_{26} C_{27} \\
\hat{Y}_3 &= \hat{C}_{30} C_{31} C_{32} C_{33} C_{34} C_{35} C_{36} C_{37} \\
\hat{Y}_4 &= \hat{C}_{40} C_{41} C_{42} C_{43} C_{44} C_{45} C_{46} C_{47} \\
\hat{Y}_5 &= \hat{C}_{50} C_{51} C_{52} C_{53} C_{54} C_{55} C_{56} C_{57} \\
\hat{Y}_6 &= \hat{C}_{60} C_{61} C_{62} C_{63} C_{64} C_{65} C_{66} C_{67} \\
\hat{Y}_7 &= \hat{C}_{70} C_{71} C_{72} C_{73} C_{74} C_{75} C_{76} C_{77}
\end{align*}
\]

\[
C_{ij} = k_i \cos \left( \frac{(2j + 1)x_i \times p}{2 \times 8} \right), \quad k_i = \begin{cases} \sqrt{1/8}, & i = 0 \\ \sqrt{2/8}, & i = 1 \ldots 7 \end{cases}
\]
Matrix for 1D 8-point DCT

• Example Use:
  For 1D DCT on a 2D Image, apply 1D horizontally and 1D vertically
  \[ Y_{8x1} = C_{8x8} X_{8x1} \quad , \quad X_{8x1} = C_{8x8}^T Y_{8x1} \]

• Matrix representation for a 2D 8x8 DCT
  • DCT: \[ Y_{8x8} = C_{8x8} X_{8x1} C_{8x8}^T \]
  • IDCT: \[ X_{8x8} = C_{8x8}^T Y_{8x1} C_{8x8} \]
IDCT Use in MPEG4 Imaging

- Execute 2D-IDCT as individual 1D-IDCTs

\[ 2D \text{ IDCT} = \text{1D IDCT} \]

One Operation

- Or design special hardware + FSL and take IDCT columns in parallel and transpose result

\[ 2D \text{ IDCT} = \text{Transpose} \rightarrow \text{1D IDCT} \]

Parallel Operations
Another Application Example: Baseline H.264 Video Encoding

I: Intra frame: DCT is used to reduce spatial redundancy within a frame.

P: Predicted frame: Motion compensated prediction - to reduce temporal redundancy. DCT is used to reduce spatial redundancy in the prediction error.
H.264 Algorithm in Cellular Phones

Source image data, 8x8 blocks

Encode Frame  Motion Estimation

DCT-based Encoder

DCT  Quantizer  Encoder

IDCT-based Encoder

IDCT  IQuant  Decoder

YUV In  YUV Out

Bitstream Out

Bitstream In
Thank you!