Simulating a Sequel “Camera” on FPGA

Table Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Rev** | **Date** | **Who** | **Description** |
|  |  |  |  |
|  |  |  |  |
| 1 | 20121130 | Henry | Initial, based on @ and Asif’s comments |

# Introduction

A “camera” simulator generates images for the instrument to process. Another way to think about this is that a real-time, full frame simulator makes it possible to fool the instrument SW (including the primary analysis) into thinking that it is talking to a real instrument (but it’s it is impossible to simulate the complete reality—otherwise it would not be a simulation—it is always a good idea to clarify the fidelity of the simulation depending on the need of the SW on the receiving end of the simulation). According to the [functional specification]((/depot/documents/Projects/Springfield/Design/Primary/FunctionalSpecs/FPGASimulatedCamera) Scott wrote almost 2 years ago already, it is not merely a trace file play-back device: it actually generates “frames” (in whatever sense of the term appropriate for Sequel) in real-time.

## Why do we need it?

Sequel chip and instrument will not show up for a couple of years at minimum. We need to make process on other parts of the system, so that the Gantt chart will work out. Going forward after the bring-up, ability to generate “images” derived from known ground truth will help validate the analysis algorithms as well.

## How much data do we have to pump through for Sequel?

The currently known Sequel design parameters:

* Pixel bit depth: 6 bit, 8 bit, 10 bit
* N\_DYE: 4
* N\_CAM (ZMW sensor): 2
* FPS: Frame rate. Use 95, increasing to 180 fps
* N\_ZMW: Number of ZMWs: upper bound has not yet been established on the number of ZMWs. The marketing desired lower bound is 5M, but I will calculate for the 1M and 2 M cases as well.
* N\_ROW, N\_COL: Each ZMW sensor will have 2 pixels. If I assign these 2 pixels row wise (i.e. N\_ROW 4K by N\_COL 2K) 8M pixels will get us 4M ZMWs.
* Required precision for the camera trace from FPGA to otto. Note that the currently used mu-law algorithm is LOSSY; if 8 bit mu-law compander representation is inadequate for the 2 amplitude optical signal, we will either have to modify the algorithm or increase the bit-depth.

Characterization of input/output:

1. Output from the FPGA
   1. Size(movie) = [N\_FRAME, N\_CAM, N\_ROW, N\_COL]
2. Input to the FPGA:
   1. If the FPGA will only do the camera trace to movie inversion:
      1. Size(camera trace) = [N\_FRAME, N\_ZMW, N\_CAM]
      2. Per ZMW or pixel dependent parameters to transform camera trace to frame:
         1. Size(PSF) = [N\_ZMW, PSF\_WIDTH, PSF\_HEIGHT]
         2. Size(hole coordinate) = [N\_CAM, N\_ZMW, 2]: the last index is 0: row, 1: col
         3. Size(BG pixel values) = [N\_CAM, N\_ROW, N\_COL] 🡪 constant, or random?
         4. Size(in-focus non-signal camera trace) = [N\_ZMW] 🡪 Poisson seed?
         5. Size(read noise) = [N\_CAM, N\_ROW, N\_COL]
   2. If the FPGA will also do the pulse to camera trace inversion before the trace to movie inversion:
      1. Ground truth pulse inversion
         1. Trc.h5 has PulseData/GroundTruth
            1. Dye
            2. Intensity index (stuck polymerase is modeled as a separate polymerase in the list of polymerases)
            3. T1 (start time)
            4. DT (duration)
         2. Pulse independent parameters
            1. Polymerase list (for indexing), and polymerase strength (or alternatively attenuation)
            2. Dead sample time (hard-coded to 1.0 for this design)
            3. Frame rate
            4. Number of total frames
            5. Size(Inverse spectral matrix) :

[N\_ZMW, N\_CAM, N\_DYE] if spectral matrix is per ZMW (requires storing to DRAM and playing back as fast as possible)

[N\_CAM, N\_DYE] if same spectral matrix for all ZMW (small enough to store in register) 🡨 choosing this option

* + 1. Per ZMW or pixel dependent parameters to transform camera trace to frame (see above list for trace to movie inversion), but the camera trace inputs disappear.

### Movie (raw pixels) transfer bandwidth

With 2 cameras, and 2 pixels per camera, here is the bandwidth required to send raw pixel data to the FPGA:

Table : Bandwidth required for full frame movie [MB/s]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Bit/ pixel** | **95 fps, 1M ZMW** | **95 fps, 2M ZMW** | **95 fps, 5M ZMW** | **180 fps, 1M ZMW** | **180 fps, 2M ZMW** | **180 fps, 5M ZMW** |
| 6 | 285 | 570 | 1425 | 570 | 1140 | 2850 |
| 8 | 380 | 760 | 1900 | 760 | 1520 | 3800 |
| 10 | 475 | 950 | 2375 | 950 | 1900 | 4750 |

There are at least 2 bottlenecks if we want to feed this much data to the FPGA:

1. PC to FPGA interconnect. We have experience using PCIe, but the synchronous send throughput from PC to FPGA is only 30 MB/s, and 400 MB/s in asynchronous mode.
2. The movie file read speed. As a reference, an ext3 file on PAP02/3/4 RAID10 reads at 20~30 MB/s.

### Camera trace transfer bandwidth

Table : Bandwidth required for camera trace of all holes [MB/s]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Bit/ trace** | **95 fps, 1M ZMW** | **95 fps, 2M ZMW** | **95 fps, 5M ZMW** | **180 fps, 1M ZMW** | **180 fps, 2M ZMW** | **180 fps, 5M ZMW** |
| 8 | 190 | 380 | 950 | 380 | 760 | 1,900 |
| 20 | 475 | 950 | 2,375 | 950 | 1,900 | 4,750 |
| 24 | 570 | 1,140 | 2,850 | 1,140 | 2,280 | 5,700 |

When sending a camera trace (which is per camera), we the bit depth is a concern. For a floating point representation, the minimum bit width I will consider is 20 bits. Using a mu-law compander will let us use 8 bits per camera trace, at the cost of information loss. The high bandwidth requirement is driven by the huge number of ZMWs. Sending the camera trace is a problem not only for the simulator, but also for the M2T FPGA (although not a scope of this discussion); you can see that the 400 MB/s PCIe asynchronous transfer speed from FPGA to PC will support only a few design choices (highlighted green) even when using the 8 bit mu-law compression.

### Pulse transfer bandwidth

Unlike the movie and camera traces, the size of the pulse file for a movie does not depend on the number of frames in a movie or the frame rate; it depends more the SNR qualities of the movie. For the worst case sizing of the pulse data bandwidth, I will use a rule of thumb of an order of magnitude reduction in bandwidth, because the file size reduction seen in Springfield from compressed camera trace to pulse is easily 20. This suggests (see the 8 bit/trace row in Table 3) that the asynchronous 400 MB/s PCIe link between PC to the FPGA will be more than adequate for 5M+ ZMWs even at 180 fps.

### Ground truth pulse generation time

## How can we do it?

The movie simulator is not responsible for M2T, so we only have to port the movie generation part of his code for the simulator, assuming that otto will handle reading the trace, MCD, and other relevant files and spoon feeding raw binary values to the FPGA over PCIe, as shown in this workflow diagram.

Offline

Optics / chip parameter

Chemistry parameter

Sample (genome) parameter

Sim

SCAM

Simulator FPGA

PCIe, 400 MB/s ingress, 30 (400) MB egress\*

M2T FPGA

“pixels”: N\_CAM × N\_ZMW× N\_PIXEL; Interconnect TBD

Otto

Camera trace: N\_CAM × N\_ZMW; PCIe (current capability: 400 MB/s)

If the PSF will be time invariant, we can pre-calculate the PSF once, encode it in an MCD file, and write that once to the FPGA, which should hold the constants in DRAM for the duration of a movie. If PSF is slowly time-varying, the FPGA will need 2 DRAM controllers: one to write the received coefficients while playing back from the other in real-time.

What program should feed the FPGA camera simulator is debatable. The PCIe communication is abstracted with a device file, to which you write 32 bit binary values, so the consideration will be based on ease of simulation model development, degree of integration with instrument SW and throughput capacity. The 2 candidates are: Matlab code (extension of the current code base) and otto (or some other C++ implementation of Crystal).

|  |  |  |
| --- | --- | --- |
|  | **Matlab** | **Otto\*** |
| Simulation model development friendliness | H | L |
| Integration with instrument SW (Homer, otto) | L | H |
| Throughput capacity (can it read and feed N number of camera traces over PCIe)? | L | H |

PCIe bandwidth from PC to FPGA over FPGA is currently 400 MB/s in asynchronous mode, and 30 MB/s or less in synchronous mode. Asynchronous mode introduces complication for the PC side SW that commands the FPGA, but this complication can be overcome. Even with only 600,000 ZMWs, sending 3 channels of COMPRESSED (8 bit mu-law) trace values at 95 fps eats up 176 MB/s, and 440 MB/s if uncompressed (20 bit floating point). But with better HW and some Verilog development, the asynchronous PCIe throughput can fairly confidently double the throughput to 800 MB/s (or possibly quadruple to 1.6 GB/s, but with more work).

Another possibility is to have separate FPGA simulator for each “camera” channels. If Sequel will have 2 “cameras”, this would effectively half the BW requirement over 1 PCIe channel. The cost will be additional SW (and possibly HW) complexity to keep the channels in sync.

### Generate ground truth pulse (offline, non-real-time)

### Produce frames from ground truth pulse on FPGA (real-time)

Pulse to movie inversion will be done in 3 steps:

Ground truth pulse 🡪 ground truth dye weighted trace

Ground truth dye weighted trace 🡪 camera trace

Camera trace 🡪 noisy frame

PSF, global BG signal (in-focus and out-of-focus), read noise

spectral matrix

Intensity list, exposure time, N\_FRAME

#### Invert ground truth pulse to dye weighted trace

Besides encoding the ground truth pulse without a noise, the ground truth contained in Springfield trace simulator output trace file (dataset PulseData/GroundTruth[[1]](#footnote-1)) is a much more compact representation, because of at least 2 reasons:

* A pulse description spans multiple frames
* If there is no pulse, we do not waste bytes to describe “no pulse”

A ZMW may have 0 or more pulses, as shown in this example (if there is no pulse for a frame of a ZMW, the ground truth dye weighted trace value will just be 0).

Figure : pulse to dye weighted trace

T1[0]

T1[1]

DT[1]

Base T

intensity

T1[0]

DT[0]

Base G

intensity

T1[0]

DT[0]

T1[1]

“t” pulse

intensity

GT DW

trace

To recap the idea represented in the above diagram to high level process for pulse to trace conversion for each frame is:

1. Initialize the dye weighted trace value to 0.
2. For each pulse that falls for a given frame (TBD: how to determine), add a product of pulse intensity and the fraction of the pulse time that falls within the integration time to the dye weighted trace value.

Because the trace to movie inverter requires the camera trace to be cursor ordered (strictly speaking, a trace is for a ZMW, but because ZMW has a unique upper left pixel, a camera trace also has an upper left pixel by which it can be sorted), the pulse to dye weighted trace conversion has to first initialize each frame’s DW trace at value 0, and then add the contribution for each pulse that fall on that frame. This means that even at this early stage, the operation is frame-oriented in these ways:

1. We start at frame number 0
2. For each frame (outer) and each dye (inner), we have to calculate the contribution for all pulses relevant for that frame, and THEN sum them to produce the total dye weighted trace value for that frame and dye. For real-time operation, this operation has to be complete well within a given exposure time (because there are downstream calculations that also have to be completed within the exposure time).

The model code that transforms ground truth pulse into trace is //depot/software/matlab/Simulator/TraceSimulator/SimPhotophysics/GetGTTraces.m, which is called by RunKineticsSSA for each ZMWs. Reproducing the comment for the function for convenience,

|  |
| --- |
| % Inputs: P, global parameters structure  % Traces, ground truth traces  % PM, GTREADS ground truth pulses data structure  % DeadMask, (deprecated)  % CumAnalogMatrix, Markov State Model for Bleach/Blink Photophysics  % GetGTTraces takes the ground truth pulses and lays these onto a set of  % traces. For each pulse a photophysics model of dye bleaching and dye  % blinking is run. This model runs is discrete time with P.SamplesPerFrame  % steps per frame. At the end of each frame there is a transfer period  % during which photons are not read. The legacy API defined this by a  % precomputed 'DeadMask', but this is computed explicitly in the current  % implementation |

Asif explained that photophysics model is disabled for most of the simulations, so I should also. Picking off the relevant lines of code from GetGTTraces (ignoring the code for photo-physics effect),

|  |
| --- |
| DT = getRunParameters(P, 'DT');  FrameTime = 1 / P.FramesPerSec; % AKA exposure time  % NumDeadSamples is a misnomer; it means just the opposite! Observe definition:  % = round(SimParams.DeadSampleTime\*SimParams.FramesPerSec\*SimParams.SamplesPerFrame)  % so this is equivalent to SamplesPerFrame (1 – DeadSampleTime \* FPS)  % If DeadSampleTime = 0, every sample in a frame is dead.  % If DeadSampleTime = 1, no sample in a frame is dead  DeadSampleIndex = P.SamplesPerFrame - getRunParameters(P, 'NumDeadSamples');  % Get relative intensity for each polymerase. The definition  % of PolymeraseZMWval is the deviation from full strength (attenuation).  % Q: up to how many polymerase can we have?  pos = P.PolymeraseZMWval; % attenuation  polInten = 1-min(pos,1.0);  % Map the sticks to the end of the polymerase list as a full intensity signal  polInten = [polInten 1];  % Make up a new polymerase (“stick”), and put it as the last polymerase in the pol list  PM.polymeraseNum(PM.stick==true)=length(P.PolymeraseZMWval)+1;  NFrames = getRunParameters(P,'TotalNumFrames');  for i=1:numel(PM.t1) % Loop over the ground truth pulses (for a ZMW)  % Determine StartFrame, EndFrame, AnalogType, and PolymeraseNum  StartFrame = min(size(Traces,1), ceil(PM.t1(i)\*P.FramesPerSec));  EndFrame = min(size(Traces,1), ceil((PM.t1(i)+PM.dt(i))\*P.FramesPerSec));  Analog = PM.BASE(i); % A pulse is for a given base  DeadSampleFactor = DeadSampleIndex / P.SamplesPerFrame; % (1 – DeadSampleTime\*FPS)  PolymeraseIntensity = polInten(PM.polymeraseNum(i)); % A polymerase generates a pulse    if (StartFrame == EndFrame) % single frame pulse!  % assign frame intensity to pulse fraction of frame  %  PM.pkmax(i) = DeadSampleFactor \* ...  PolymeraseIntensity \* PM.dt(i) / FrameTime; % Note: dt < exposure time  % Note: trace was initialized to 0  Traces(StartFrame,Analog) = Traces(StartFrame,Analog) + PM.pkmax(i);  else  % assign first frame intensity  % Integrate from real start time to frame boundary time  inten = DeadSampleFactor \* PolymeraseIntensity \* ...  (StartFrame - PM.t1(i)\*P.FramesPerSec);  Traces(StartFrame,Analog) = Traces(StartFrame,Analog) + inten;  PM.pkmax(i) = inten;    % assign last frame intensity  % Integration time works out to: (exposure time) + (end of frame) – real end time  inten = DeadSampleFactor \* PolymeraseIntensity \* ...  ((PM.t1(i)+PM.dt(i))\*P.FramesPerSec - EndFrame + 1);  Traces(EndFrame,Analog) = Traces(EndFrame,Analog) + inten;  PM.pkmax(i) = max(PM.pkmax(i),inten);    % if there are intermediate frames, assign these too  %  if (EndFrame-1 > StartFrame),  PM.pkmax(i) = DeadSampleFactor \* PolymeraseIntensity;  Traces(StartFrame+1:EndFrame-1,Analog) = PM.pkmax(i); % Why not increment??  end  end  end |

If I ignore dead sample factor (Austin and Asif agreed, because dead sample factor is irrelevant for CMOS camera), the code simplifies somewhat:

|  |
| --- |
| DT = getRunParameters(P, 'DT');  FrameTime = 1 / P.FramesPerSec; % AKA exposure time  % Get relative intensity for each polymerase. The definition  % of PolymeraseZMWval is the deviation from full strength (attenuation).  % Q: up to how many polymerase can we have?  pos = P.PolymeraseZMWval; % attenuation  polInten = 1-min(pos,1.0);  % Map the sticks to the end of the polymerase list as a full intensity signal  polInten = [polInten 1];  % Make up a new polymerase (“stick”), and put it as the last polymerase in the pol list  PM.polymeraseNum(PM.stick==true)=length(P.PolymeraseZMWval)+1;  NFrames = getRunParameters(P,'TotalNumFrames');  % Q: what does nume1 do?  for i=1:numel(PM.t1) % Loop over the ground truth pulses (for a ZMW)  % Determine StartFrame, EndFrame, AnalogType, and PolymeraseNum  StartFrame = min(size(Traces,1), ceil(PM.t1(i)\*P.FramesPerSec));  EndFrame = min(size(Traces,1), ceil((PM.t1(i)+PM.dt(i))\*P.FramesPerSec));  Analog = PM.BASE(i); % A pulse is for a given base  PolymeraseIntensity = polInten(PM.polymeraseNum(i)); % A polymerase generates a pulse    if (StartFrame == EndFrame) % single frame pulse!  % assign frame intensity to pulse fraction of frame  %  PM.pkmax(i) = PolymeraseIntensity \* PM.dt(i) / FrameTime; % Note: dt < exposure time  % Note: trace was initialized to 0, so just increment  Traces(StartFrame,Analog) = Traces(StartFrame,Analog) + PM.pkmax(i);  else  % assign start frame intensity  % Integrate from real start time to frame boundary time  inten = PolymeraseIntensity \* (StartFrame - PM.t1(i)\*P.FramesPerSec);  Traces(StartFrame,Analog) = Traces(StartFrame,Analog) + inten; % increment  PM.pkmax(i) = inten;    % assign end frame intensity  % Integration time works out to: (exposure time) + (end of frame) – real end time  inten = PolymeraseIntensity \* ...  ((PM.t1(i)+PM.dt(i))\*P.FramesPerSec - EndFrame + 1);  Traces(EndFrame,Analog) = Traces(EndFrame,Analog) + inten; % increment end frame  % take the maximum of the pkmid so far for the end frame  PM.pkmax(i) = max(PM.pkmax(i),inten);    if (EndFrame-1 > StartFrame), % intermediate frames  PM.pkmax(i) = PolymeraseIntensity;  Traces(StartFrame+1:EndFrame-1,Analog) = PM.pkmax(i); % replace?  end  end  end |

#### Invert ground truth dye weighted trace to camera trace

The input into the 2nd step is the ground truth dye weighted trace, produced at N\_ZMW \* FPS per second. As shown in Table 3, this works out to 4.75 GB/s (somewhat irrelevant unless it has to be stored in a memory device).

The forward (camera to dye) spectral matrix is N\_DYE×N\_CAM, which is not necessarily square. The inverse (dye to camera) matrix is therefore N\_CAM×N\_DYE. One complication is that the matrix is position interpolated during camera trace to dye weighted trace conversion (during T2P), but for the simplicity of FPGA implementation, I will just use the same matrix for all holes, avoiding both the complication of position interpolation and the need to play back a per-ZMW spectral matrix from DRAM (for N\_DYE = 4 and N\_CAM = 2, this would have worked out to 8 floats per ZMW, or 40M floats for N\_ZMW = 5M; at 180 fps and 20 bit floating points, this would have required a DRAM read speed of 18 GB/s, much greater than the DRAM read speed capacity of 6.4 GB/s on ML605 evaluation board).

TODO: add in-focus non-signals to the ground truth camera trace?

#### Invert camera trace to pixels

The movie generation part of the Matlab code (in //depot/software/matlab/Simulator/TraceSimulator/SimMovieToTrace folder) is

function [mov,v]=simCameraSubFrames(camTraces,psfMatrix,globalBG,readNoise, ...

                                    inFocusBaseline,inFocusOnly)

, which applies the in-focus, out of focus, and shot noise in just a few lines of code:

|  |
| --- |
| [ny,nx,nPSFs] = size(psfMatrix);  [nFrames,nZMWs] = size(camTraces);  if (nZMWs < nPSFs)      fprintf('warning: replicating %.1f%% of ground-truth traces\n',nPSFs/nZMWs\*100);  end  mov = zeros(ny,nx,nFrames);  v = zeros(ny,nx);  for i=1:nPSFs      g = repmat(camTraces(:,rem(i-1,nZMWs)+1)',[nx\*ny 1]);      g = reshape(g,[ny nx nFrames]);  % mov += psfMatrx .\* camTrace, essentially, but accumulate neighboring pixels      mov = mov + bsxfun(@times,psfMatrix(:,:,i),g);  % v += inFocusBaseline .\* psfMatrx, essentially, but accumulate neighboring pixels      v = v + inFocusBaseline(rem(i-1,nZMWs)+1) .\* psfMatrix(:,:,i);  end  mov = mov + globalBG; % Add out-of-focus signal  %  % photoelectron shot noise -  for speed and memory usage use gaussian  % approximation above muThr value  %  muThr = 30;     % qqplot(poissrnd(muThr,1,1e6)) check  iLo = mov < muThr; % Which pixel values are lower than muThr?  mov(iLo) = single(poissrnd(double(mov(iLo)))); % Replace those with POISS(lambda=pixel)  % Otherwise (above threshold), add ( SQRT(pixel) \* RAND(mean=0, std=1) )  mov(~iLo) = mov(~iLo) + randn(size(mov(~iLo))) .\* sqrt(mov(~iLo));  %  % gaussian read-noise: add for all pixels  %  mov = mov + single(randn(size(mov)) \* readNoise);  % Irrelevant for camera simulator, since it only produces raw pixel values  % noise model variance: inFocuseBaseline .\* PSF MX + globalBG + RN^2  %  v = v + globalBG + readNoise^2; |

psfMatrix is formulated upstream, where psfs are arrayed according to the layout and interpolated from a library. This contains each isolated psf in full image but in separate planes. It is debatable whether that operation has a real-time requirement; it will depend on how rapidly the PSF changes in reality. Note that even though the current Springfield otto CAN update the PSF and the hole locations during acquisition, it does NOT do that. Repeated experiments have shown that the drift control keeps the holes in the starting location on the data camera sufficiently well, to rule out the need to update the hole locations. The same may not be true for the PSF, but has never been proved either way, but there has never been a reason to suspect that the PSF changes drastically over the course of a movie. For Sequel, we have to revisit both assumptions.

# Ground truth pulse generator detailed design

The truth simulator writes a time invariant PSF for each ZMW.

# PC SW (SCAM) detailed design

# FPGA Implementation detailed design

For each ZMW, PSF is represented as a matrix of size (PSF\_WIDTH, PSF\_HEIGHT) whose upper left “pixel” is given by the (start\_row, l\_col) coordinate. This PSF representation is similar to the PSF in a weighted summer FPGA design, except for dropping the assumption that a PSF is square. Because the PSF from neighboring ZMW can overlap, a pixel intensity for a camera is a super-composition of multiple camera trace for that camera, as you can see in the rendering below.

ZMW 0

ZMW 1

ZMW 2

ZMW 3

ZMW 4

ZMW 5

ZMW 1234

ZMW 1235

To implement an engine that projects the camera traces to a pixel in real-time, an inverse of the PSF is required; until a better name is found, I will just call this FSP (reverse of PSF). To use minimum amount of FPGA resource necessary, the FSP have to be cursor ordered (this is similar to the ordering of the PSF in weighted summer design). In the weighted summer design, the fixed size of a PSF allowed a row reducer that began at some column start\_col and deterministically stopped at (start\_col + PATCH\_SIZE – 1); that is, a reduced row weighted sum always picked up the contribution from PATCH\_SIZE number of columns (although some of the weights were 0). Still, I want to borrow from the weighted summer design to the extent possible. Still there will be unavoidable differences.

|  |  |
| --- | --- |
| **M2T weighted sum (reducer) FPGA design** | **T2M (expander) design** |
| Does not require that the design starts working at the upper left corner of a frame. | Design starts working with ZMW 0 from the get-go. That is, a design that multiplies a FSP matrix to camera traces requires traces to generate a pixel (of course), so even for pixels that are deliberately dissociated from any ZMW, the design requires a fake ZMW to generate those pixels. |
| Requires just 1 row reducer per row to be available at each clock cycle. This is possible because the ZMWs are spaced apart by regular pixel distances (PATCH\_PITCH). For the same reason, only 1 answer (reduced sum) is generated for each row each clock cycle. | Because an FSP size varies for each pixel (the minimum being 1×1—only 1 ZWM contributes to the pixel), multiple “expander” (inverse of the reducer) may be required in the same clock cycle, and multiple expander may produce the completed pixel value in the same clock cycle. Together, this means that the number of pixels produced by the design will not be constant in time. |

# References

Springfield movie simulator is used extensively for the 1mary analysis training. Perforce location of simulator component code base: //depot/software/matlab/Simulator.

* [Sharepoint folder](http://sharepoint/rd/Springfield/Forms/AllItems.aspx?RootFolder=%2Frd%2FSpringfield%2FSystems%20Engineering%2FSimulatorSoftwareDesign&InitialTabId=Ribbon%2EDocument&VisibilityContext=WSSTabPersistence) containing simulator software design documents
* [ER diagram](http://sharepoint/rd/Springfield/Systems%20Engineering/SimulatorSoftwareDesign/SimulationSoftwareArchitecture.graphml) of simulator components (open with yEd, [www.yworks.com/en/products\_**yed**\_about.html](http://www.yworks.com/en/products_yed_about.html) ).

One thing that concerns me right away: in the high level ER diagram, there is no path from Enzyme kinetics file to mov.h5 file (output of the MovieSimulator), let alone reference genome.  It appears that only the optics/ photonics calibration input files drive the MovieSimulator.

Gt.xml 🡪 EnzymeSim 🡪 gt.h5

mi.xml 🡪 Optics Sim (Hans wrote Sequel model) 🡪 mi.h5 ()

tsim.xml 🡪 TraceSim 🡪 trc.h5 (TraceData, PulseData/GroundTruth dataset, ScanData—makes MCD unnecessary)

1. Used widely in 1mary analysis (such as T2P traning, PulseError tool; see \\depot\\software\dotnet\PacBio\Analysis\Data\SimPulseReader.cs for example) [↑](#footnote-ref-1)