Tutorial: Zynq Implementation of Subset + Arm-Learning Algorithm

Introduction

In this lab, we will use Vivado High Level Synthesis (HLS) and Software Development Kit (SDK) to create a peripheral capable of executing the subset + arm-learning algorithm using ARM Cortex-A9 processor system on Zynq. We will use Vivado IPI to create a top-level design, which includes the Zynq processor system as a sub-module.

GitHub Links:

https://github.com/shubhrajit19196/ASLonZynqFloat

https://github.com/shubhrajit19196/ASLonZynqFixed

Design Description

The Zynq implementation of any algorithm involves hardware-software codesign. The fundamental steps of the subset + arm-learning algorithm (for N = 4 and K = 2) along with the hardware-software partitioning is represented below:

Step I: For the first 11 (total number of subsets) time slots, select all the possible subsets sequentially. Otherwise, select the subset as determined by step VIII. (Processor)

Step II: Play the selected subset and determine whether among the arms of the selected subset, the number of 'occupied arms' is less than or equal to K = 2. (Processor)

Step III: If step II gives true, then update the Xa and Ta values and the learnt probabilities for the arms of the selected subset. Otherwise, do not update the Xa and Ta values of any arm. (FPGA)

Step IV: Calculate the learned probabilities of all the 4 arms as Xa/Ta. (FPGA)

Step V: In the background, play all the subsets which have one or more arms in common with the current selected subset and update their Xs, Ts and Ns values. (FPGA)

Step VI: While playing a subset in the background, the status of the common arms is known but the status of the remaining arms is determined by comparing a random number against the learnt probabilities of those arms. (FPGA)

Step VII: Calculate the Q-value for all the subsets. (FPGA)

Step VIII: Determine the subset having the highest Q-value and move to step I. (FPGA)

General Flow for this Lab

→ Generate Vivado IPs using Vivado HLS.

→ Integrate all the IPs with the Zynq Processing System by creating a block diagram in Vivado.

→ Generate bitstream in Vivado.

→ Generate baremetal application in Vivado SDK.

→ Test the design.

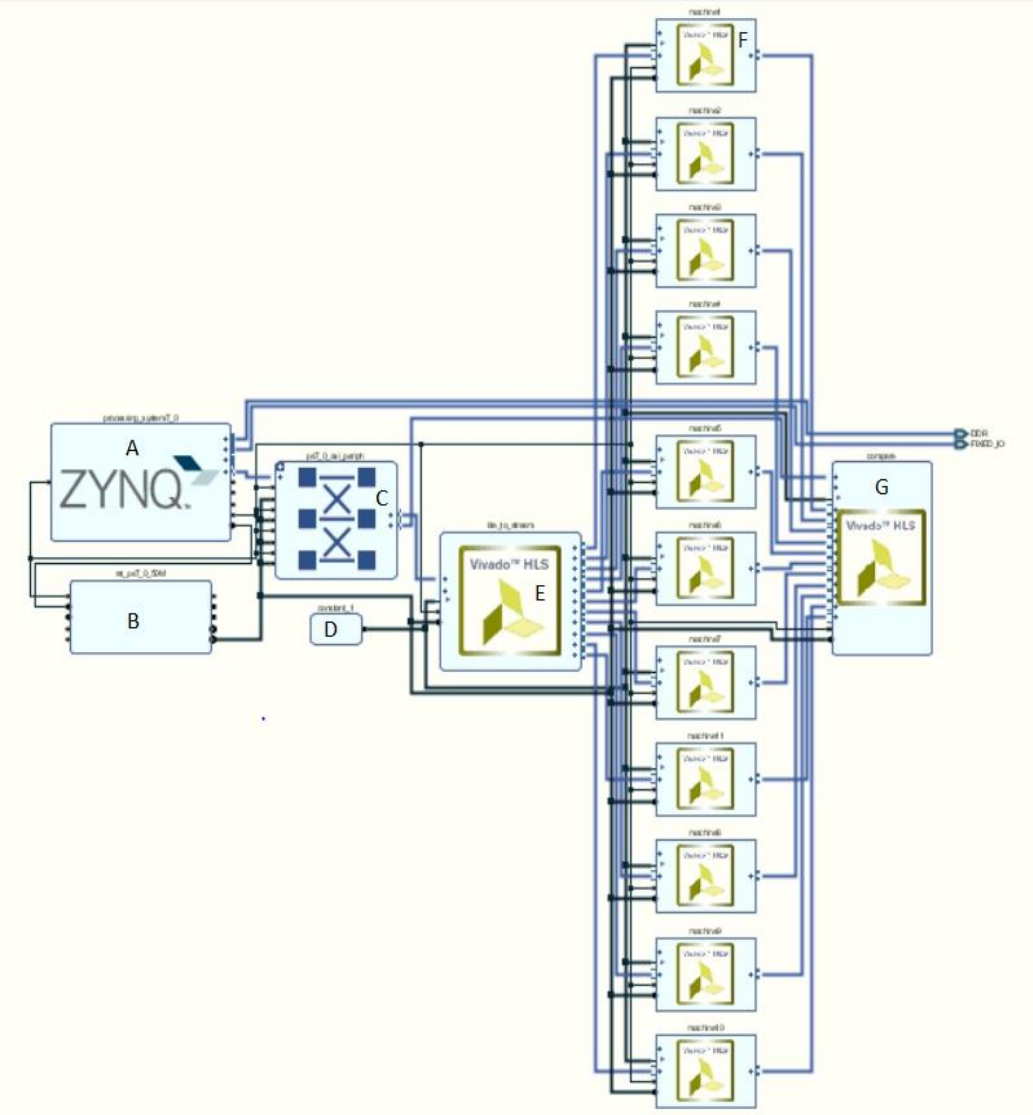

Figure 1. The system block design

Procedure

Step 1. Generating the machine IP (marked as F)

- Create a new project in **Vivado HLS.**
- Select **ZC706** as the board and set clock period as **20 ns**.
- Add a top file and a test file **having extension .cpp** (for example, top.cpp and test.cpp)

- In the top file, write the C code to **initialize and update the variables Xs, Ts and Ns** of the machine as per the inform signal received from the lite_to_stream IP via **AXI stream protocol**. Then **calculate the Q value** (quality-factor) of the machine according to the **UCB (Upper Confidence Bound)** formula and **pass the calculated Q-value** to the comparator IP via **AXI stream protocol.**
- **Write the testbench** in the test file (say test.cpp).
- Perform **HLS C simulation.**
- Perform **HLS C synthesis.**
- Perform **HLS C/RTL cosimulation.**
- Perform **HLS export RTL.**

Step 2. Generating the comparator IP (marked as G)

- Create a new project in **Vivado HLS.**
- Select **ZC706** as the board and set clock period as **20 ns**.
- Add a top file and a test file **having extension .cpp** (for example, top.cpp and test.cpp)
- In the top file, write the C code to **compare the 11 Q-values** received from the 11 machine IPs via **AXI stream protocol** and **pass the index of the highest Q-value machine** to the processor via **AXI lite protocol.**
- **Write the testbench** in the test file (say test.cpp).
- Perform **HLS C simulation.**
- Perform **HLS C synthesis.**
- Perform **HLS C/RTL cosimulation.**
- Perform **HLS export RTL.**

Step 3. Generating the lite_to_stream IP (marked as E)

- Create a new project in **Vivado HLS.**
- Select **ZC706** as the board and set clock period as **20 ns**.
- Add a top file and a test file **having extension .cpp** (for example, top.cpp and test.cpp)
- In the top file, write the C code to **receive the 9-bit inform signal** from the processor via **AXI lite protocol**, **calculate the learned probabilities** of the 4 arms, **play the 4 arms by comparing the learned probabilities against a random number generated by a PRNG** (pseudo-random number generator) and **pass 11 different 18-bit inform signals** to the 11 machine IPs via **AXI stream protocol.**
- **Write the testbench** in the test file (say test.cpp).
- Perform **HLS C simulation.**
- Perform **HLS C synthesis.**
- Perform **HLS C/RTL cosimulation.**
- Perform **HLS export RTL.**

Step 4. Draw the block diagram

- Create a new project in **Vivado.**
- Go to **Settings → IP → Repository → Add the three generated HLS IPs to the repository** one by one.

- Click on **Create Block Design**. Give any suitable name to the block design (say design_asl).
- Click on **Add IP** or press **ctrl + I** to add each IP to the block design.
- Add the **ZYNQ7 Processing System** (marked as A). Click on **Run block automation** pop-up.
- Add the **lite_to_stream** IP (marked as E). Click on **Run connection automation** pop-up.
- After this the **AXI Interconnect** (marked as C) and the **Processor System Reset** (marked as B) will get created automatically.
- Add 11 instances of the **machine** IP (marked as F)**.** Click on **Run connection automation** pop-up.
- Add the **comparator** IP (marked as G). Click on **Run connection automation** pop-up.
- Add the **Constant** IP (marked as D). It always gives a constant 1-bit value of 1.
- Connect this constant 1 value to the **ap_start** port under the **ap_ctrl** drop-down of all the HLS IPs.
- **Make all the remaining connections** as shown in the block diagram.
- Click on **Validate Design.**
- Click on **Save Block Design.**

Step 5. Generate bitstream

- Right-click on the block design name present under the **Sources** tab and click on **Create HDL Wrapper.** Then select **Let Vivado manage wrapper and auto-update.**
- Again right-click on the block design name present under the **Sources** tab and click on **Generate Output Products…**
- Click on **Generate Bitstream.**

Step 6. Developing the baremetal application

- Go to **File → Export → Export Hardware.**
- Tick the checkbox **Include bitstream** and click **OK.**
- Go to **File → Launch SDK.**
- In the SDK window, go to **File → New → Application Project.**
- Give any suitable name to the project (say test_asl). Click on **Next → Select Hello World → Finish.**
- Write the C code to **run the algorithm for some given number of time slots, say N = 10000.**
- In each time slot, one AXI read and one AXI write will take place to **send the inform signal to the FPGA** and to **receive the highest Q-value subset index from the FPGA**.
- In each time slot, the **selected subset is played by comparing a random number to the probability statistics of the 4 arms** that has been considered and the **status of the trial is sent back to the FPGA via the inform signal**.
- Store an internal copy of the **X, T and N values for all the 11 subsets** in the SDK and also **keep adding the reward for each time slot** to obtain the total reward after N time slots for display purpose.

- **Calculate the total time taken to run all the N time slots** of the algorithm by capturing the time at the beginning and at the end of the algorithm by using the XTime_GetTime() function.

Step 7. Running the baremetal application

- Click on **Program FPGA.**
- Open any terminal **(say TeraTerm)**. Select **Serial** and click **OK**.
- Go to **Setup → Serial port… → Select Speed as 115200 →** click **OK.** Here 115200 is the **UART Baud Rate.**
- Click on **Run As → Launch on Hardware (GDB).**
- The results of execution of the algorithm will be printed on the terminal.

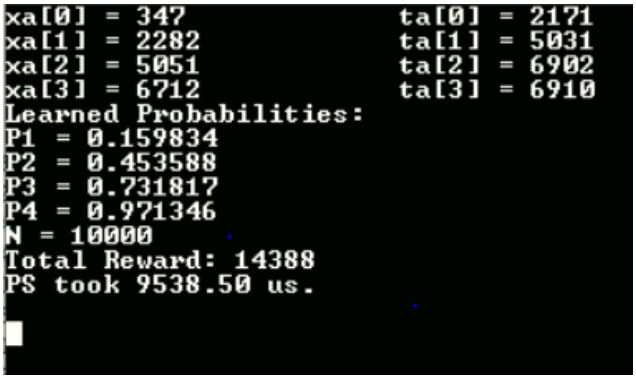

Figure 2. Snapshot of sample output on the terminal.

8. Enabling Neon coprocessor optimization in SDK

- Right-click on the name of the project under the **Project Explorer** tab.
- Go to **C/C++ Build Settings → Optimization → Set Optimization Level as Optimize most (-O3).**
- Then follow the sub-steps given in step 7 to run the application with coprocessor optimization enabled.

Note: For running the entire algorithm on ARM processor, write the C code for the entire algorithm in the SDK itself and then run according to the sub-steps given in step 7 (the program FPGA step is omitted). Also for enabling Neon coprocessor optimization, follow the sub-steps given in step 8.

Results

Consider four different types of probability statistics:

PS1 = [0.1, 0.3, 0.5, 0.7]

PS2 = [0.51, 0.52, 0.53, 0.54]

PS3 = [0.11, 0.21, 0.31, 0.41]

PS4 = [0.38, 0.24, 0.77, 0.65]

A) For standard-precision floating-point, SP-FL:

Performance comparison for 10000 time slots (for PS1)

Algorithm	ARM + FPGA	ARM + FPGA + NEON	ARM + NEON	ARM Only
Subset+Arm-learning	9537.59 us	8486.24 us	52267.98 us	147952.68 us

Efficiency of the algorithm in terms of reward obtained in 10000 time slots

Algorithm	Reward for PS1	Reward for PS2	Reward for PS3	Reward for PS4
Subset+Arm-learning	14355	18669	8623	18681

Resource utilization

Algorithm	Slice LUTs	Slice Registers	F7 Muxes	DSPs	Block RAM Tile	Bonded IOPADS	BUFGCTRL
Subset+Arm Learning	48022	24644	275	201	8	130	1

Power Consumption: 1.965 W

B) For fixed word-length, WL = 21, FL = 7:

Performance comparison for 10000 time slots (for PS1)

Algorithm	ARM + FPGA	ARM + FPGA + NEON	ARM + NEON	ARM Only
Subset+Arm-learning	9531.57 us	8738.82 us	52267.98 us	147952.68 us

Efficiency of the algorithm in terms of reward obtained in 10000 time slots

Algorithm	Reward for PS1	Reward for PS2	Reward for PS3	Reward for PS4
Subset+Arm-learning	14394	18780	8461	18649

Resource utilization

Algorithm	Slice LUTs	Slice Registers	F7 Muxes	DSPs	Block RAM Tile	Bonded IOPADS	BUFGCTRL
Subset+Arm Learning	10916	7842	142	50	13.5	130	1

Power consumption: 1.641 W