Tutorial: Zynq Implementation of 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-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/ALonZynqFloat

https://github.com/shubhrajit19196/ALonZynqFixed

Design Description

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

Step I: Toggle between selecting (1,2) or (3,4) as the subset for the first N/K = 2 time slots. Otherwise, select the top β arms from the sorted array of arm indices as determined in step V and step VI. (Processor)

Step II: Play the selected arms and determine whether the number of 'occupied arms' is more than K = 2. (Processor)

Step III: If step II gives true, then no reward is added to the X value any arm. Otherwise, the reward is added to the X values of the selected arms. The T values of the selected arms are incremented by 1 irrespective of the condition defined in step II. (Processor)

Step IV: Calculate the learned probabilities and the Q-values for all the 4 arms. (FPGA)

Step V: Sort the indices of the arms with respect to their Q-values in descending order. (FPGA)

Step VI: Determine β by performing some arithmetic calculations using the learned probabilities 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.

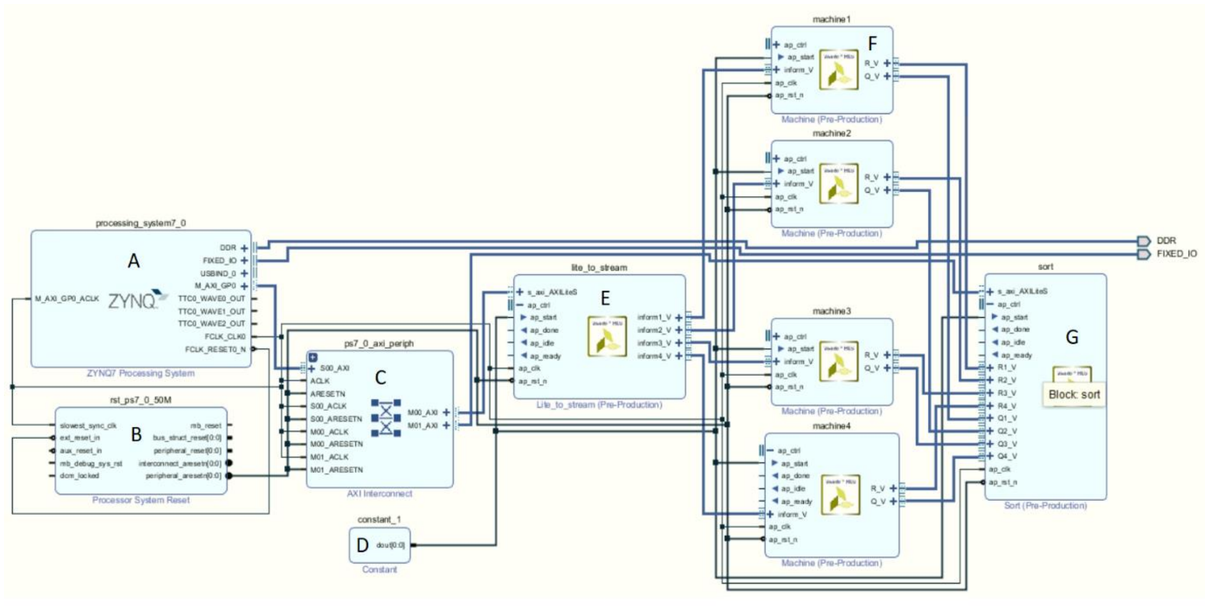


Figure 1. The system block design

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 X, T and N** of the machine as per the inform signal received from the lite_to_stream IP via **AXI stream protocol**. Then **calculate the learned probability (X/T) and the Q value** (quality-factor) of the machine according to the **UCB (Upper Confidence Bound)** formula and **pass the calculated learned probability and Q-value** to the sort 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 sort 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 **sort (in descending order) the indices of the 4 arms with respect to the 4 Q-values** received from the 4 machine IPs via **AXI stream protocol.** Then **calculate β by performing some arithmetic calculations using the 4 learned probabilities** received from the 4 machine IPs via **AXI stream protocol.** Then

pass the sorted arm indices and **the calculated β value** 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** and **pass 4 different 3-bit inform signals** to the 4 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_al).
- 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 4 instances of the **machine** IP (marked as F)**.** Click on **Run connection automation** pop-up.
- Add the **sort** 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

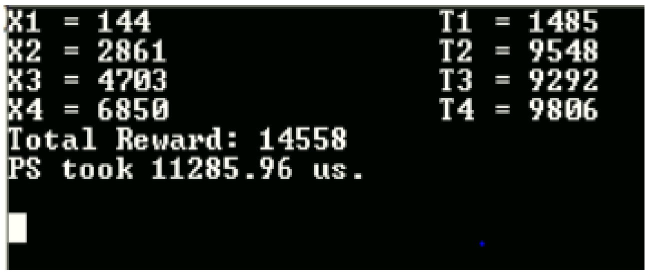
- Right-click on the block design name present under the **Sources** tab and click on **Create HDL Wrapper.** Then select **Let Vivado manage wrapper and auto-update.**
- Again right-click on the block design name present under the **Sources** tab and click on **Generate Output Products…**
- Click on **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_al). 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 sorted arm indices and the β value from the FPGA**.
- In each time slot, the **top β arms from the sorted array of arm indices are 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**.
- **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.



```
X1 = 144                    T1 = 1485
X2 = 2861                   T2 = 9548
X3 = 4703                   T3 = 9292
X4 = 6850                   T4 = 9806
Total Reward: 14558
PS took 11285.96 us.
```

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
Arm-learning	11275.11 us	8627.21 us	11488.17 us	30456.17 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
Arm-learning	15093	21160	9275	18186

Resource utilization

Algorithm	Slice LUTs	Slice Registers	F7 Muxes	DSPs	Block RAM Tile	Bonded IOPADS	BUFGCTRL
Arm-Learning	14816	9709	84	73	-	130	1

Power Consumption: 1.705 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
Arm-learning	11352.48 us	8616.52 us	11488.17 us	30456.17 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
Arm-learning	14572	20404	8850	17744

Resource utilization

Algorithm	Slice LUTs	Slice Registers	F7 Muxes	DSPs	Block RAM Tile	Bonded IOPADS	BUFGCTRL
Arm-Learning	3549	2985	4	25	2	130	1

Power consumption: 1.605 W