Vitis Code Tutorial

Ahmet Can Mert
ahmet.mert@iaik.tugraz.at

TU
Graz

Overview

- Structure of Vitis_code:

```
├── main.cc             : Main file calling test function.
├── lscript.ld          : It sets the stack/heap size of the processor. Do NOT modify.
├── README.txt          : -
├── Xilinx.spec         : -
├── communication.c     : It has FPGA-CPU communication related functions. You do NOT have to make any changes on this file.
├── communication.h     : -
├── instruction.c       : It has functions for generating INS array (i.e., instructions encoded with address, OPCODE etc.) for different operations.
├── instruction.h       : -
├── platform.c          : It has FPGA platform related functions. You do NOT have to make any changes on this file.
├── platform.h          : -
├── platform_config.h: -
└── pke
    ├── modular_arithmetic.cc
    ├── modular_arithmetic.h
    ├── pke_parameters.h
    ├── poly_arithmetic2.c
    ├── poly_arithmetic2.h
    ├── randombytes.c
    ├── randombytes.h
    ├── test_polymul.c
    └── test_polymul.h
```

main.cc

- This is the main file that Vitis runs. It performs some initilizations first and then calls the function `test_polymul()` which is defined in `test_polymul.c/test_polymul.h` in `pke` folder. You can keep this file the same and just modify `test_polymul()`.

```
init_platform();
//axi_address_base   = (uint32_t *) 0x00A0000000;    // For zcu102 board
axi_address_base   = (uint32_t *) 0x40000000;        // For PYNQ-z2 board

int TEST_TYPE = 0; // 0: TRNG, 1:AES, 2:PKE, 3:End

printf("******************************************************************\n");
printf("*                          Starting SDK                          *\n");
printf("******************************************************************\n");


while(TEST_TYPE !=3){
    if(TEST_TYPE==0)        test_polymul();
    else break;

    printf("Type of test [0: PolyMul, 3:End] : ");
    scanf("%d", &TEST_TYPE);
}
```

pke/test_polymul.c

- It has two functions: `test_polymul()` and `poly_mult_HW()`.
  - `test_polymul()` function generates two polynomials of size 256, a and b, and then performs coefficient-wise multiplication of a and b using HW and SW. HW operation is performed by `poly_mult_HW()` function.

  - `poly_mult_HW()` function presents an example for executing operations on FPGA. It performs the following steps:
    - Sends data to FPGA
    - Encodes instructions for the cryptoprocessor and sends instructions to FPGA
    - Executes the instructions on FPGA
    - Reads data back from the FPGA

    For the Task-3 of your assignment, you can either modify `poly_mult_HW()` or write a new function similar to `poly_mult_HW()`.

poly_mult_HW() function

- Now, we'll look into poly_mult_HW() function in detail.
  - Sending data to FPGA.

```
uint32_t i,j;
uint64_t INS[64];
uint64_t fpga_memory_data[1024]; // This is a memory buffer in SW.

// send data to  FPGA.
for(i=0; i<256; i++){
    fpga_memory_data[i] = data1[i];
    fpga_memory_data[i+256] = data2[i];
}
for(i=512; i<1024; i++)
    fpga_memory_data[i] = 0;

send64(fpga_memory_data, 0, 1024, 0);   // Send the buffer to FPGA's BRAM
delay(100);
```

We define two arrays of 64-bit data.

- INS has size of 64 and it is used to send instructions to instruction memory of the cryptoprocessor.
- fpga_memory_data has size of 1024 and it is used to send 64-bit coefficients to the data memory of the cryptoprocessor.

Here, we set the coefficients 0-255 to the first polynomial, coefficients 256-511 to the second polynomial and the rest is 0.

```
send64(uint64_t *p, uint32_t base_address, uint32_t num_words, uint32_t INS_flag);
```

function (defined in communication.c) sends data to the FPGA from CPU. It takes 4 inputs.

- *p: array of 64-bit data that you want to send to FPGA
- base_address: starting write address for data/instruction memory of the cryptoprocessor
- num_words: number of 64-bit data (word) that you want to send to FPGA
- INS_flag: If this is 0, you are sending coefficients to the data memory of the cryptoprocessor
  If this is 1, you are sending instructions to the instruction memory if the cryptoprocessor

For this example, we're sending 1024 words in fpga_memory_array to the data memory of the cryptoprocessor starting from address 0 of the data memory of the cryptoprocessor.

poly_mult_HW() function

- Now, we'll look into `poly_mult_HW()` function in detail.
  - Encoding instructions for the cryptoprocessor and sends instructions to FPGA.

```
// INS: instruction array, Operand1 address=0, Operand2 address=256, Output result address=512.
init_INS_Poly_mult(INS, 0, 256, 512);

send64(INS, 0, 64, 1); // send polynomial multiplication INStruction to FPGA using send64() INS_flag=1.
delay(100);
```

For this example, we're sending whole INS array to the program memory of the cryptoprocessor starting from address 0 of the program memory. The content of INS is already set by `init_INS_Poly_mult()`.

`init_INS_Poly_mult()` function (defined in `instruction.c`) takes 3 input operands (starting address of operand 1 polynomial in data memory, starting address of operand 2 polynomial in data memory and starting address of resulting polynomial in data memory), encodes these operands with instruction code=24. Please note that this function is specific to the instruction instruction code =24. You can create a similar function for every instruction (and instruction code) that you define. In `instruction.c`, you can find some other examples as well.

```
void init_INS_Poly_mult(unsigned long long *INS, unsigned int INP_ADDRESS1, unsigned int INP_ADDRESS2, unsigned int OUT_ADDRESS)
{
    int i;
        printf("OUT_ADDRESS=%d\n", OUT_ADDRESS);


        unsigned long long program_word =  ((unsigned long long) 1<<35) + ((unsigned long long) OUT_ADDRESS<<25) + (INP_ADDRESS2<<15)+(INP_ADDRESS1<<5) + 24;

        //printf("Program word = %llu\n", program_word);
        unsigned long long ins_group[64] =
        {
                0x0000000000,
                0x0830000014,   // This instruction is overwritten by the input
                0x0800000000,
                0x0000000000,
                0x000000001f    // End of computation
        };

        ins_group[1] = program_word;
        for(i=0; i<64; i++)
            INS[i] = ins_group[i];
}
```

Generating instruction. instruction code is 24. If you want to modify this function or create similar one, only this part should be changed (i.e., use your instruction code instead of 24).

No need to modify this part.

`poly_mult_HW()` function

- Now, we'll look into `poly_mult_HW()` function in detail.
  - Executing instructions on the FPGA and reading data back to CPU.

```
exeIns();  // Now ask the FPGA to compute the instruction (i.e., polynomial multiplication)
delay(100);

receive64(fpga_memory_data, 0, 1024); // Read the BRAM of FPGA into the SW-side buffer
delay(100);
```

`exeIns()` function (defined in `communication.c`) sends commands to the FPGA for performing the instructions. You do NOT have to modify this function.

```
receive64(uint64_t *p, uint32_t base_address, uint32_t num_words);
```

function reads data from the data memory of the cryptoprocessor.
It takes 3 inputs:

- *p: array of 64-bit data that will store the incoming data from FPGA
- base_address: starting read address for data memory of the cryptoprocessor
- num_words: number of 64-bit data (word) that you want to read from the FPGA

For this example, it reads 1024 coefficients from data memory of the cryptoprocessor into the array `fpga_data_memory`, starting from address 0 of the data memory of the cryptoprocessor.

Other files in pke folder

- In pke folder, we provide some functions that you can use.
  - `pke_parameters.h`: You can ignore this file.
  - `poly_arithmetic2.c`: Empty file, just provides template for implementing schoolbook method.
  - `randombytes.c`: Includes a function to generate random byte.
  - `modular_arithmetic.cc`: Includes field arithmetic functions (integer addition/subtraction and modular multiplier) for 64-bit integers. Note that modular multiplier method is NOT using Montgomery method.

- You can use these functions (or you can write your own functions) to implement schoolbook/Karatsuba functions (in case you want to implement this approach and perform a part of operation in SW).