Study and development of RISC-V based application

EC036 Darshil Modha

EC056 Piyush Saini

EC078 Tithi Shah

Objective:

- Our main goal is to make an application on a microcontroller which was created using this new architecture, RISC-V.
- This goal needs to be in-line with the limitations of the microcontroller and simulation alike.
- Current plan is to make a protocol converter, as to which protocols it will support is yet to be decided; we are exploring several different ideas.

PHASE-1

RISC-V

- RISC-V is an open standard instruction architecture based on the RISC principles. Unlike other processors that follow the CISC architecture, for example, 80386, RISC architecture only has a small number of opcodes at the base level.
- RISC-V is one such architecture that allows the user to add their own commands to the instruction set because of its open-source nature. Since, most CPU manufacturers like Intel, AMD, ARM, etc. require monetary compensation in return for access to their design ideologies, it becomes extremely difficult academically to learn about microprocessors. RISC-V was made with the goal of academic usability in mind; as it grew, big companies started getting interested in it. Like many RISC designs, RISC-V is also a load-store architecture.

RISC-V ISA (Instruction Set Architecture):

- A RISC-V ISA is defined as a base integer ISA, which must be present in any implementation, plus optional extensions to the base ISA. A base is carefully restricted to a minimal set of instructions sufficient to provide a reasonable target for compilers, assemblers, linkers and operating systems, and so provides a convenient ISA and software toolchain “skeleton” around which more customized processor ISAs can be built.
- There are two primary base integer variants, RV32I and RV64I, which provide 32-bit and 64-bit address spaces respectively. RV32E is a subset variant of the RV32I base instruction set, mainly used for small microcontrollers, and which has half the number of integer registers. And a future variant of 128-bit address space RV128I which has not been implemented on any hardware as of now. One thing to keep in mind is that RV32I is distinct from RV64I and is not a subset.

Simulators explored:

- Freedom Studio by SiFive
  - Problem: Only has a single microcontroller support which is made by SiFive. Also this is under development so it has many bugs which haven't been solved yet. Debugging requires hardware, no direct simulation support.
- Embedded Studio by Segger
  - Problem: Overall less amount of debugging features, no access to SoC registers.
- Andesight by Andestech
  - This is the simulator which was chosen as the appropriate one
  - Reason: Trial version is available for 90 days with full access to the software and allows creation of programs for premade microcontrollers and microprocessors.

Microcontroller:

- Since we were able to get access to the Andesight IDE, we are going to use one of their many available microprocessor: NX25(F)
- Although NX25(F) itself is only a processor, the IDE has peripheral support which will allow us to build our application.

NX25(F) Features:

• 5-stage in-order execution pipeline

• Hardware multiplier– radix-2/radix-4/radix-16/radix-256/fast

• Hardware divider

• Optional branch prediction: 4-entry return address stack (RAS)

– Choice of

* Static branch prediction, or

* Dynamic branch prediction

· 32/64/128/256-entry branch target buffer (BTB)

· 256-entry branch history table

· 8-bit global branch history

NX25(F) Features (cont’d):

- Machine mode and optional User mode
- Optional performance monitors
- Misaligned memory accesses
- RISC-V physical memory protection
- RV64I Base instruction set
- Extensions to the base include:
  - C (compressed instruction), M (integer multiplication), A (atomic instructions/bit manipulation), N (user level interrupt and exception handling), F & D (single/double-precision floating-point instructions).

The processor seems like an overkill. Why use this one?

- Since this area of technology is new to us, there was not much forethought regarding what type of processor to use, we just chose whatever was available to us and was supported by the IDE.
- Surprisingly, this processor as the manufacturer claims is used in communication systems, where data communication protocols are used; our application is about protocol conversion so it fits nicely into the entire objective.

Code for a basic blinking LED (assembly language)

- Assembly code includes:
    - Accessing GPIO registers through its address
    - Setting direction of GPIO register as output
    - Loop to blink the led using the data out register
    - Note: Start up is done using simple C code, external assembly function is called from main().
    - Purpose of this code:
        - To get a basic understanding of RISC-V assembly code
        - To test the GPIO pins on the microcontroller

Problems faced: (1st phase)

- Reference manual to the mentioned microcontroller wasn’t readily available but technical support from Andestech provided one upon request.
- Since we’ve been suggested to not use assembly language by our mentor we are looking for ways to access the GPIO structures defined in the header file (similar to what we’ve learnt in the subject of embedded systems) through C program.
- Platform specific header files are available within the software but we haven’t found a way to include them in any project. The compiler cannot link the source and header files together even when gpio header files are in the same workspace directory.
- An application definition cannot be formed right now because of the above mentioned reasons

PHASE-2

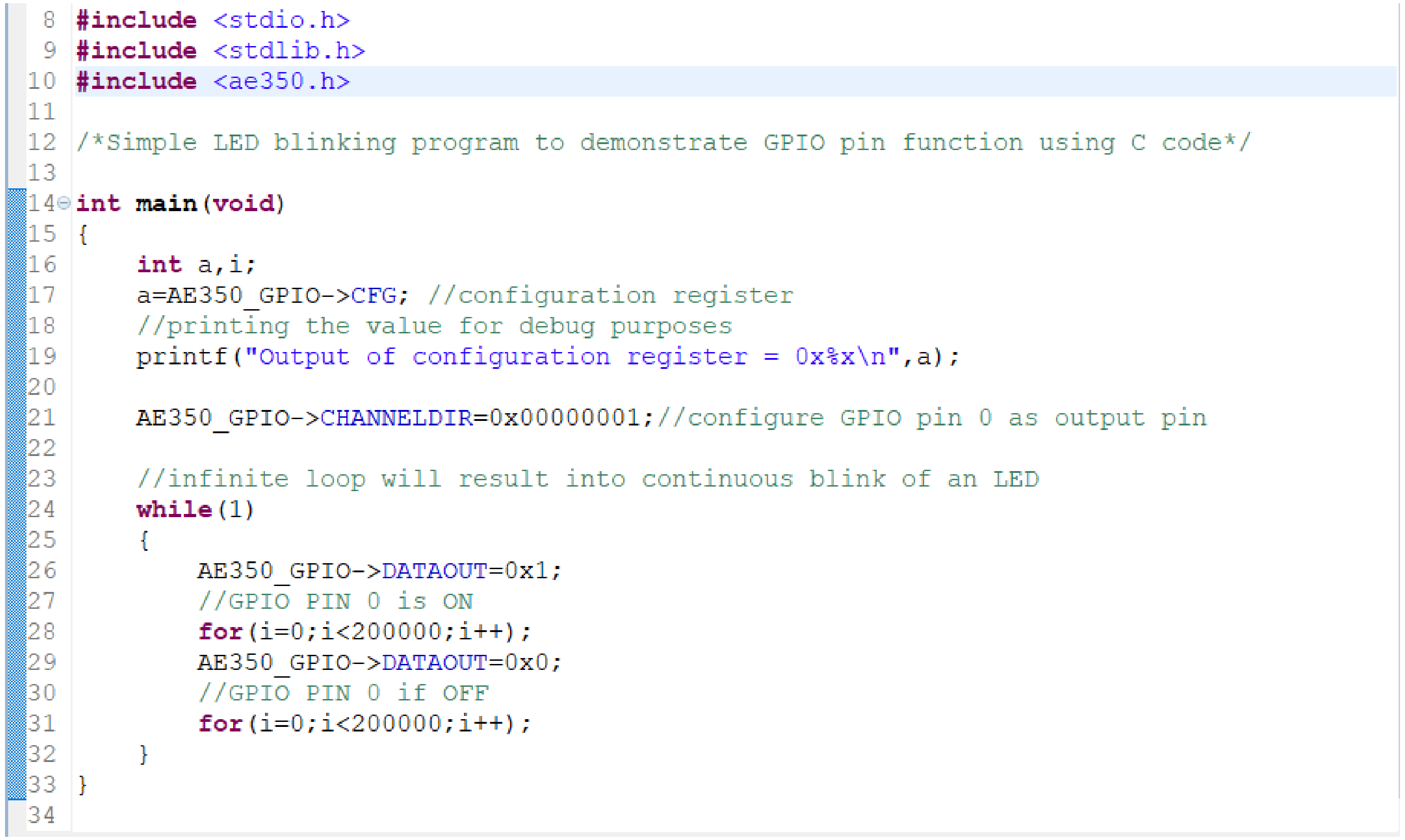
Problems solved:

- We were able to write LED blink program using C.
- Platform specific header files are now included in projects as required.
- Plan is to make a protocol converter circuit, I2C-to-SPI and such.

How did we do it?

- Fortunately, tech support personnel replied and we were able to find the required header files for the microcontroller. **(Header files are available at this location: $ANDESIGHT_ROOT->amsi->bsp)**
- Example of a header file: (stm32f10x.h)
- Now, to include these files we changed the build settings for the *project* ***(Your_project_name->right-click->build settings->Compiler Settings->Include directories, same thing needs to be done for Linker settings as well and that option can be found right below compiler settings).***
- We created a common folder with all the necessary header files and then appended the path to those in the aforementioned field.

Code for LED blink using C only:



```c
#include <stdio.h>
#include <stdlib.h>
#include <ae350.h>

/*Simple LED blinking program to demonstrate GPIO pin function using C code*/

int main(void)
{
    int a,i;
    a=AE350_GPIO->CFG; //configuration register
    //printing the value for debug purposes
    printf("Output of configuration register = 0x%x\n",a);

    AE350_GPIO->CHANNELDIR=0x00000001;//configure GPIO pin 0 as output pin

    //infinite loop will result into continuous blink of an LED
    while(1)
    {
        AE350_GPIO->DATAOUT=0x1;
        //GPIO PIN 0 is ON
        for(i=0;i<200000;i++);
        AE350_GPIO->DATAOUT=0x0;
        //GPIO PIN 0 if OFF
        for(i=0;i<200000;i++);
    }
}

```

Code (cont’d):

- Purpose:
  - To learn how to use structures and user defined data types included in the header files.
  - Basic understanding of GPIO used in the IDE ATGCPIO100.

Problems: (2nd phase)

- Which protocols to be used is still to be figured out. Moreover, how much support of protocol simulation is available on the IDE needs to be determined. (Refer Virtual Registers in Keilv5 for example of simulating a slave/master on software).

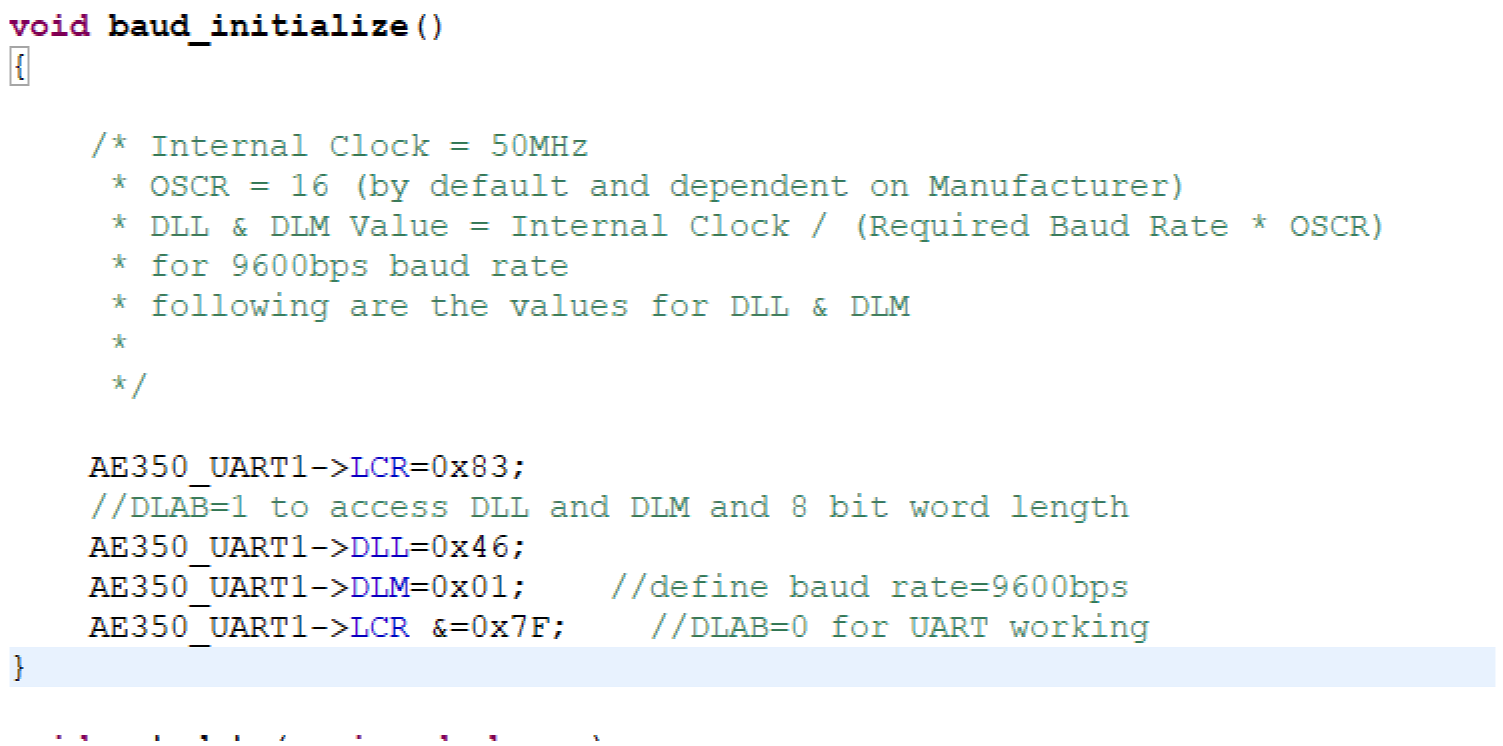
Plans: (2nd Phase, Week 2)

- At least simulate I2C, SPI and a simple UART.
- Decide which two protocols will be used for the converter and figure out an approach for it.

Simulation of I^{2}C and SPI

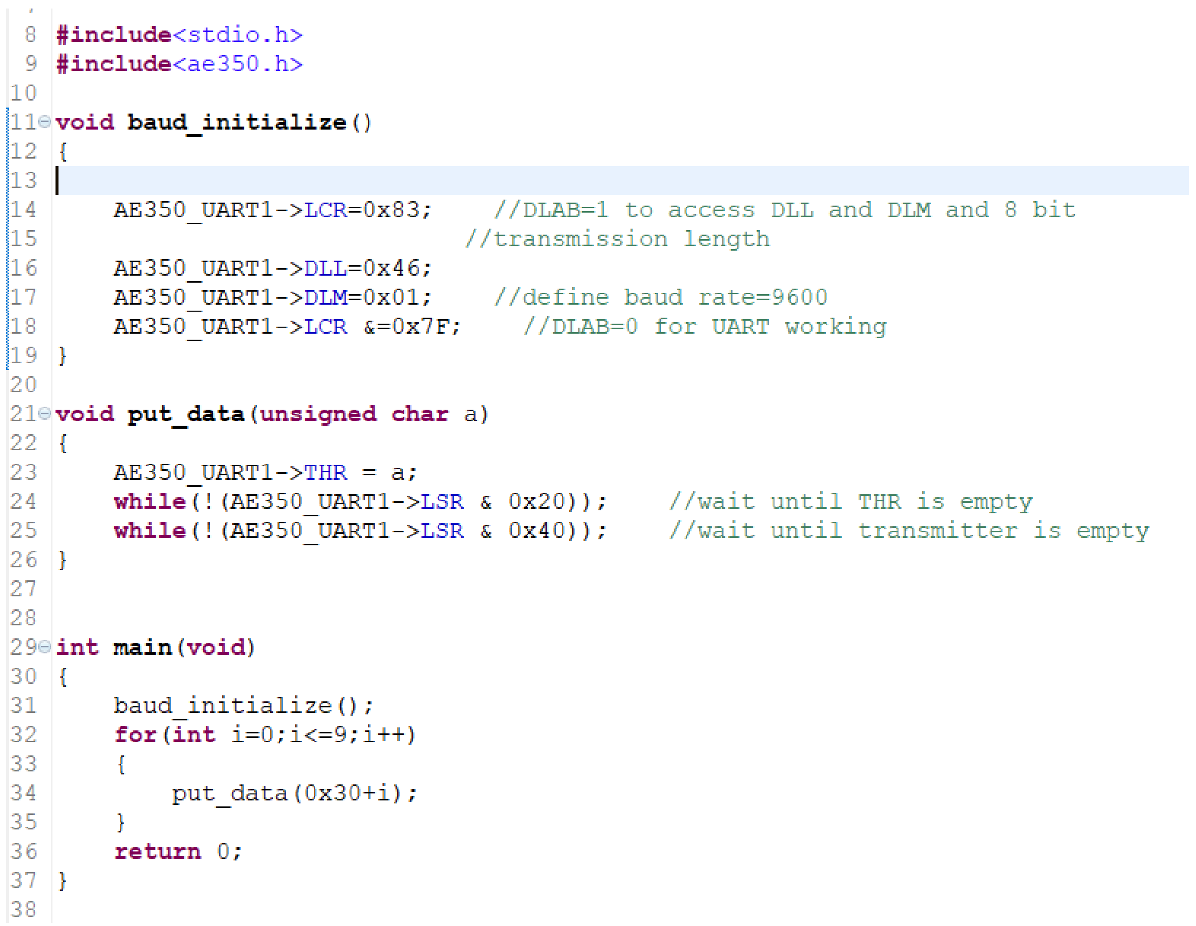
- Since no simulation support is given for these two protocols i.e. we weren’t able to do anything with it.

Simulation of UART: Initialization



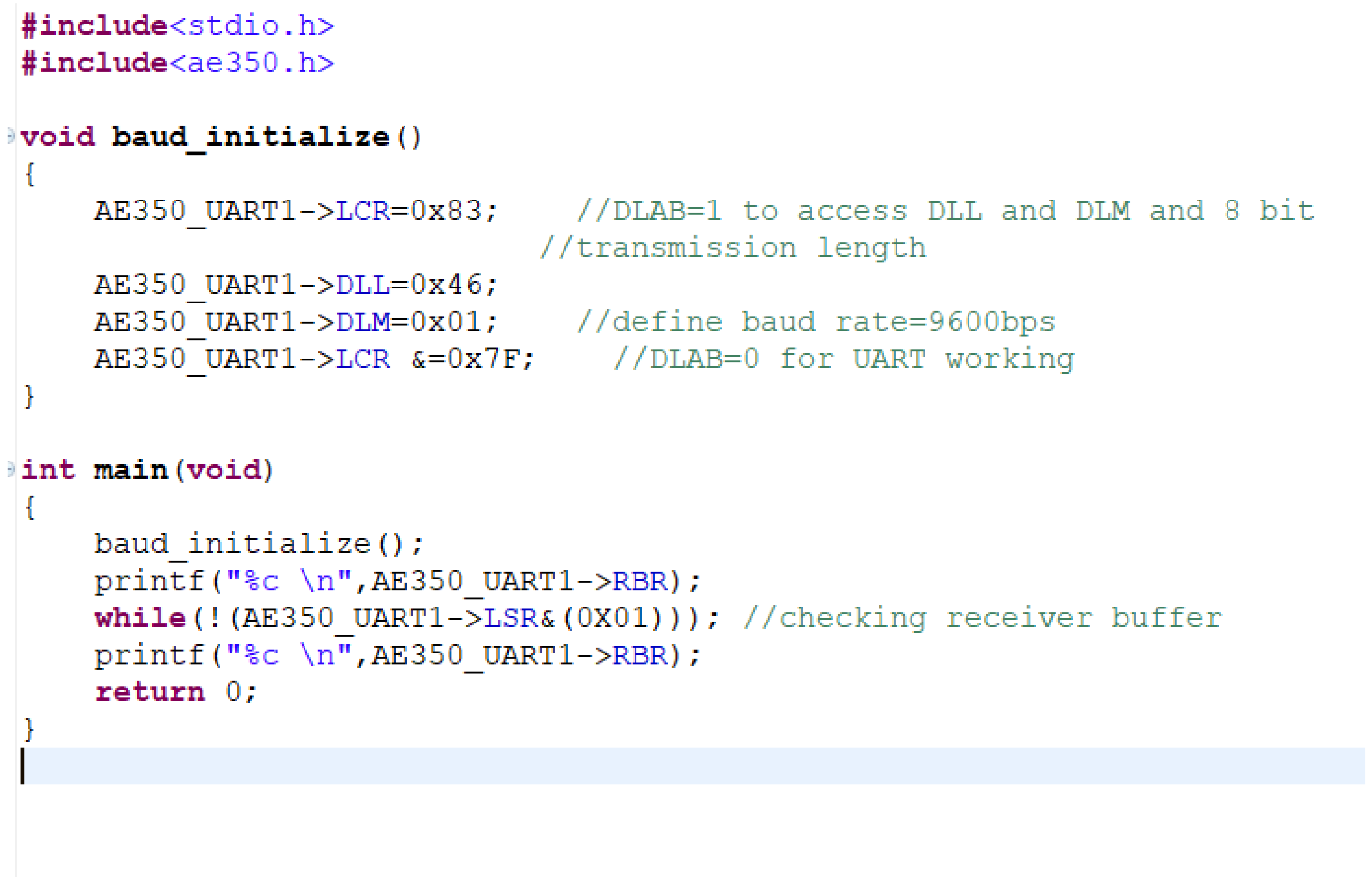
```
void baud_initialize()
{

    /* Internal Clock = 50MHz
     * OSCR = 16 (by default and dependent on Manufacturer)
     * DLL & DLM Value = Internal Clock / (Required Baud Rate * OSCR)
     * for 9600bps baud rate
     * following are the values for DLL & DLM
     *
     */

    AE350_UART1->LCR=0x83;
    //DLAB=1 to access DLL and DLM and 8 bit word length
    AE350_UART1->DLL=0x46;
    AE350_UART1->DLM=0x01;    //define baud rate=9600bps
    AE350_UART1->LCR &=0x7F;    //DLAB=0 for UART working
}
```

Simulation of UART: (Tx only)



```
#include<stdio.h>
#include<ae350.h>

void baud_initialize()
{

    AE350_UART1->LCR=0x83;     //DLAB=1 to access DLL and DLM and 8 bit
                             //transmission length
    AE350_UART1->DLL=0x46;
    AE350_UART1->DLM=0x01;     //define baud rate=9600
    AE350_UART1->LCR &=0x7F;     //DLAB=0 for UART working
}

void put_data(unsigned char a)
{
    AE350_UART1->THR = a;
    while(!(AE350_UART1->LSR & 0x20));     //wait until THR is empty
    while(!(AE350_UART1->LSR & 0x40));     //wait until transmitter is empty
}


int main(void)
{
    baud_initialize();
    for(int i=0;i<=9;i++)
    {
        put_data(0x30+i);
    }
    return 0;
}

```

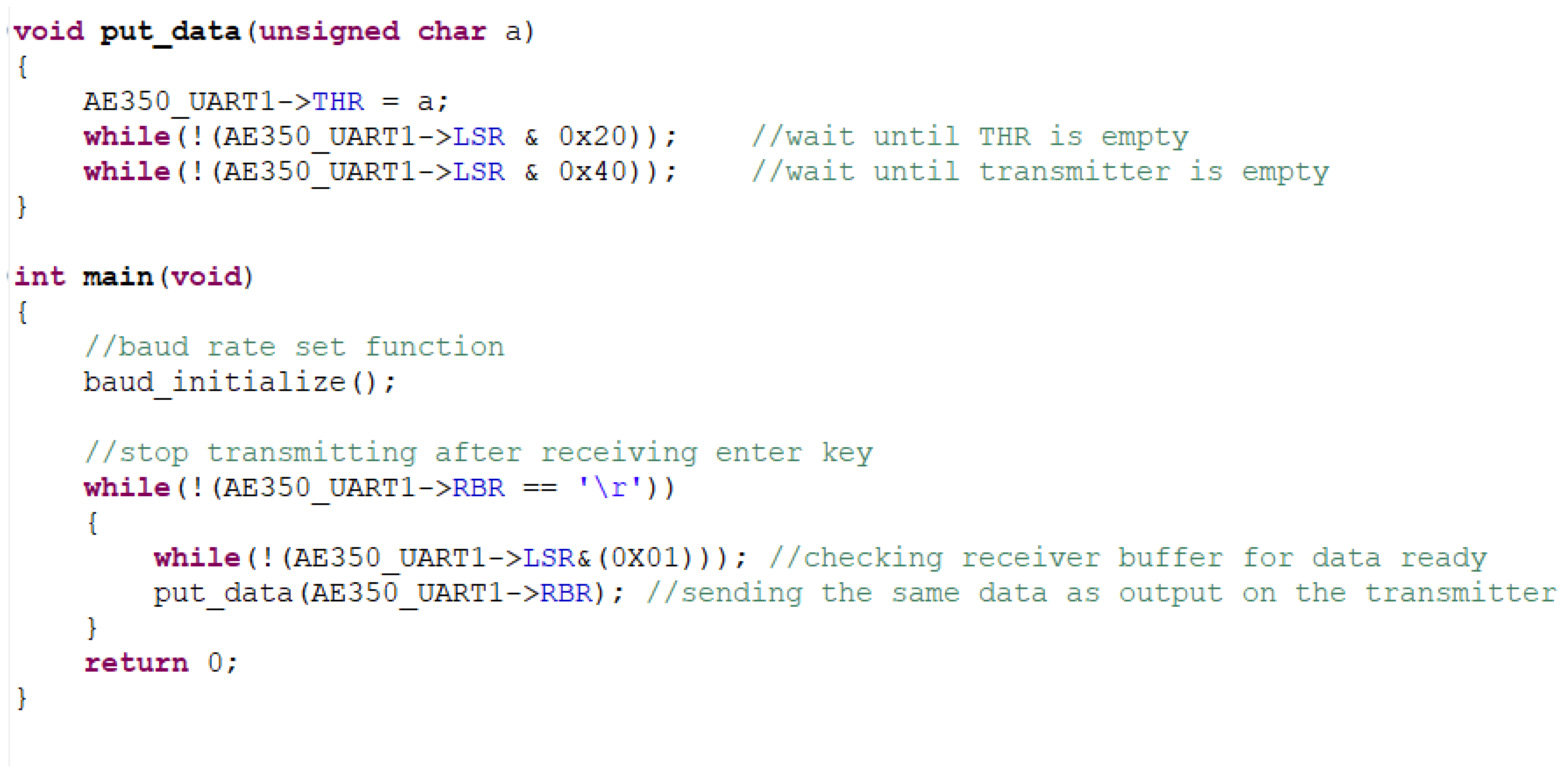
Simulation of UART: (Rx only)



```
#include<stdio.h>
#include<ae350.h>

void baud_initialize()
{
    AE350_UART1->LCR=0x83;    //DLAB=1 to access DLL and DLM and 8 bit
                             //transmission length
    AE350_UART1->DLL=0x46;
    AE350_UART1->DLM=0x01;    //define baud rate=9600bps
    AE350_UART1->LCR &=0x7F;    //DLAB=0 for UART working
}

int main(void)
{
    baud_initialize();
    printf("%c \n",AE350_UART1->RBR);
    while(!(AE350_UART1->LSR&(0X01))); //checking receiver buffer
    printf("%c \n",AE350_UART1->RBR);
    return 0;
}
```

Simulation of UART: (Tx and Rx, full duplex)

Initialize function is same as before



```
void put_data(unsigned char a)
{
    AE350_UART1->THR = a;
    while(!(AE350_UART1->LSR & 0x20));    //wait until THR is empty
    while(!(AE350_UART1->LSR & 0x40));    //wait until transmitter is empty
}

int main(void)
{
    //baud rate set function
    baud_initialize();

    //stop transmitting after receiving enter key
    while(!(AE350_UART1->RBR == '\r'))
    {
        while(!(AE350_UART1->LSR&(0X01))); //checking receiver buffer for data ready
        put_data(AE350_UART1->RBR); //sending the same data as output on the transmitter
    }
    return 0;
}
```

Problems faced:

- No proper way to simulate peripherals such as I2C, SPI and RTC.
- Our previous idea of protocol conversion can’t be implemented.
- A new application is to be found.

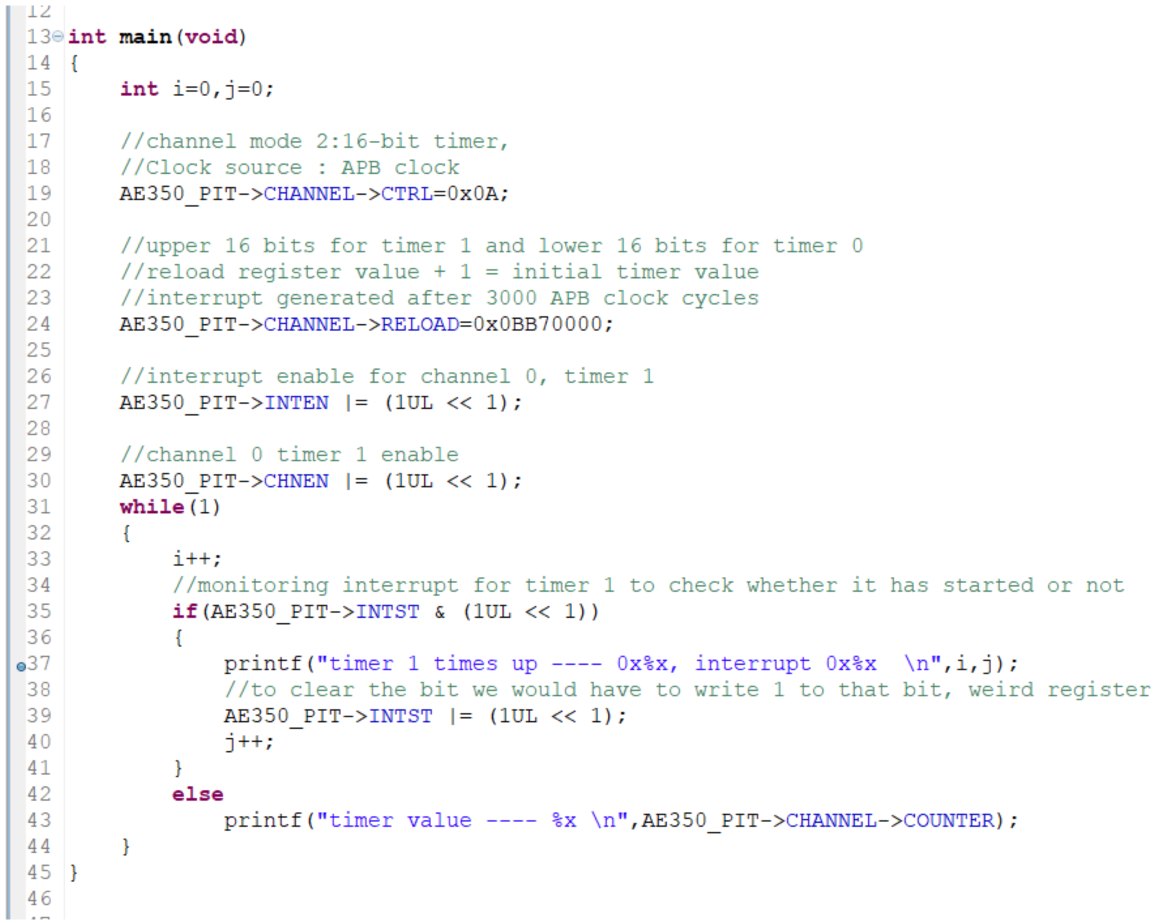
Changes in Objective:

- Since, we can only simulate GPIO and UART, there is no way to make a protocol converter.
- We have also explored Real Time Clock as there was simulation support available for it. However, the simulated clock is not able to synchronize with internal clock and output is glitchy and unreliable.
- Limitations of this simulation call for a change in the objective of the project.
- Final application will only include UART and GPIO, and maybe a few interrupts if we are able to simulate those.

Programmable Interrupt Timer:

- Due to simulation limitations, we explored other aspects of the microcontroller in search of a different objective.
- Programmable interrupt timer or PIT is a collection of multifunction timers which can be used as pulse width modulators and simple timers.
- AMBA (advanced microprocessor bus architecture) support
- 4 multi-function timers
- Programmable source clock
- External Pause functions

Timer program to show functionality:



```c
int main(void)
{
    int i=0,j=0;

    //channel mode 2:16-bit timer,
    //Clock source : APB clock
    AE350_PIT->CHANNEL->CTRL=0x0A;

    //upper 16 bits for timer 1 and lower 16 bits for timer 0
    //reload register value + 1 = initial timer value
    //interrupt generated after 3000 APB clock cycles
    AE350_PIT->CHANNEL->RELOAD=0x0BB70000;

    //interrupt enable for channel 0, timer 1
    AE350_PIT->INTEN |= (1UL << 1);

    //channel 0 timer 1 enable
    AE350_PIT->CHNEN |= (1UL << 1);
    while(1)
    {
        i++;
        //monitoring interrupt for timer 1 to check whether it has started or not
        if(AE350_PIT->INTST & (1UL << 1))
        {
            printf("timer 1 times up ---- 0x%x, interrupt 0x%x  \n",i,j);
            //to clear the bit we would have to write 1 to that bit, weird register
            AE350_PIT->INTST |= (1UL << 1);
            j++;
        }
        else
            printf("timer value ---- %x \n",AE350_PIT->CHANNEL->COUNTER);
    }
}
```

Phase-3

Case study:

- Incorporation of all the features supported by the software into 1 program.
- The controller will shift from one mode to other through user input.
- Interrupts not included because of many bugs.

Problems:

- Timer behavior is abnormal (PIT) – not possible to disable channel.
- Note: We have submitted this bug to tech support and they have recorded it.
- Configuration files for interrupts is not global and specific to only demo programs.

References:

- AndeSight_STD_V3.2_USER_MANUAL
- ATGCGPIO100_N25_REFERENCE_MANUAL
- RISC-V instruction manual downloaded from RISC-V GitHub page (privileged and unprivileged sets)
- AndeSight_STD_v3.2_User_Manual_UM186_V1.5