SİVAS UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF ENGINEERING AND NATURAL SCIENCES

MICROPROCESSORS

Experiments Manual

Supervisor: Asst. Prof. Nurhan Güneş
Asst. Prof. Recep Emir

Prepared by: Res. Asst. Berke Can Turan
Res. Asst. Beyza Demirkoparan
Res. Asst. Şekip Dalgaç
Res. Asst. İremnur Duru
Res. Asst. Metehan Arı

SİVAS

TABLE OF CONTENTS

1. Experiment 1 in 8086 Emulator: Displaying 'Hello World' Using BIOS Interrupt in 8086 Assembly

Objective of the Experiment

To display a 'Hello World' message on the screen using the BIOS video interrupt INT 10h in 8086 assembly language. The experiment introduces interrupt-driven output and segment:offset addressing.

- Use BIOS interrupts for video output
- Manage segment:offset memory addressing (ES:BP)
- Understand low-level string output using INT 10h
- Appreciate control and precision in interrupt-based programming

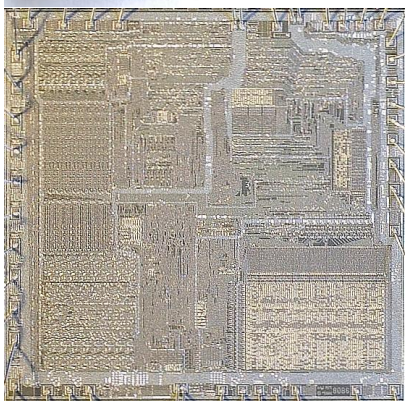
Theoretical Background

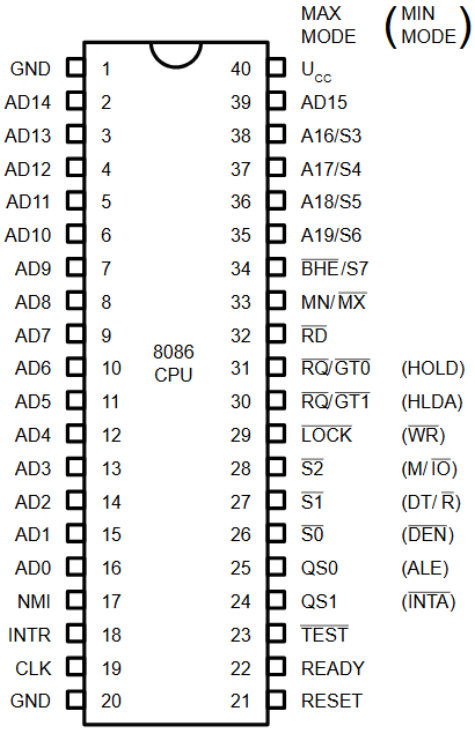
A Microprocessor is a programmable device that takes in input, performs some arithmetic and logical operations over it, and produces the desired output. In simple words, a Microprocessor is a digital device on a chip that can fetch instructions from memory, decode and execute them, and give results. It is an important part of a computer architecture without which you will not be able to perform anything on your computer.

A Microprocessor takes a bunch of instructions in machine language and executes them, telling the processor what it has to do. The microprocessor performs three basic things while executing the instruction:

- It performs some basic operations like addition, subtraction, multiplication, division, and some logical operations using its Arithmetic and Logical Unit (ALU). New Microprocessors also perform operations on floating-point numbers.
- Data in microprocessors can move from one location to another.
- It has a Program Counter (PC) register that stores the address of the next instruction based on the value of the PC, Microprocessor jumps from one location to another and makes decisions.

The 8086 microprocessor is an 8-bit/16-bit microprocessor designed by Intel in the late 1970s. It is the first member of the x86 family of microprocessors, which includes many popular CPUs used in personal computers.





The architecture of the 8086 microprocessor is based on a complex instruction set computer (CISC) architecture, which means that it supports a wide range of instructions, many of which can perform multiple operations in a single instruction. The 8086 microprocessor has a 20-bit address bus, which can address up to 1 MB of memory, and a 16-bit data bus, which can transfer data between the microprocessor and memory or I/O devices.

The 8086 microprocessor has a segmented memory architecture, which means that memory is divided into segments that are addressed using both a segment register and an offset. The segment register points to the start of a segment, while the offset specifies the location of a specific byte within the segment. This allows the 8086 microprocessor to access large amounts of memory, while still using a 16-bit data bus.

The 8086 microprocessor has two main execution units: the execution unit (EU) and the bus interface unit (BIU). The BIU is responsible for fetching instructions from memory and decoding them, while the EU executes the instructions. The BIU also manages data transfer between the microprocessor and memory or I/O devices.

The 8086 microprocessor has a rich set of registers, including general-purpose registers, segment registers, and special registers. The general-purpose registers can be used to store data and perform arithmetic and logical operations, while the segment registers are used to address memory segments. The special registers include the flags register, which stores status information about the result of the previous operation, and the instruction pointer (IP), which points to the next instruction to be executed.

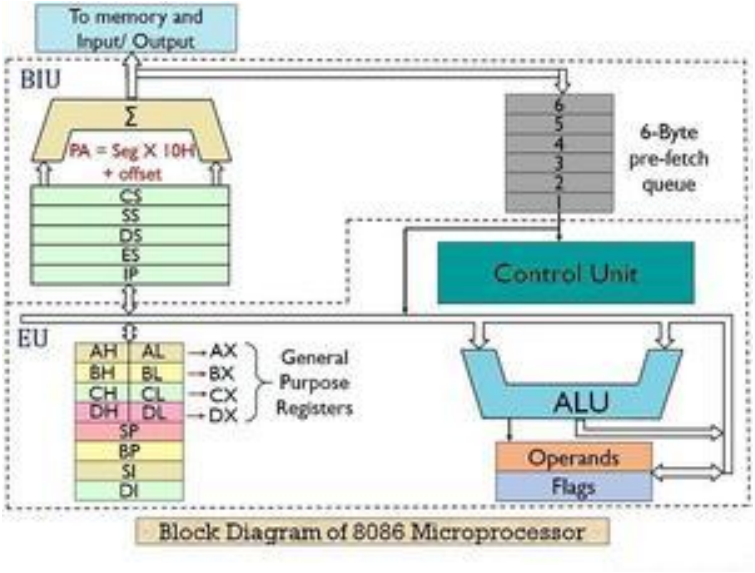
A Microprocessor is an Integrated Circuit with all the functions of a CPU. However, it cannot be used stand-alone since unlike a microcontroller it has no memory or peripherals.

8086 does not have a RAM or ROM inside it. However, it has internal registers for storing intermediate and final results and interfaces with memory located outside it through the System Bus.

In the case of 8086, it is a 16-bit Integer processor in a 40-pin, Dual Inline Packaged IC.

The size of the internal registers(present within the chip) indicates how much information the processor can operate on at a time (in this case 16-bit registers) and how it moves data around internally within the chip, sometimes also referred to as the internal data bus.

8086 provides the programmer with 14 internal registers, each of 16 bits or 2 bytes wide. The main advantage of the 8086 microprocessor is that it supports Pipelining.



Block Diagram of 8086 Microprocessor

Memory segmentation:

- In order to increase execution speed and fetching speed, 8086 segments the memory.
- Its 20-bit address bus can address 1MB of memory, it segments it into 16 64kB segments.
- 8086 works only with four 64KB segments within the whole 1MB memory.

The internal architecture of Intel 8086 is divided into 2 units: The Bus Interface Unit (BIU), and The Execution Unit (EU). These are explained as following below.

The Bus Interface Unit (BIU):
It provides the interface of 8086 to external memory and I/O devices via the System Bus. It performs various machine cycles such as memory read, I/O read, etc. to transfer data between memory and I/O devices.

BIU performs the following functions are as follows:

- It generates the 20-bit physical address for memory access.
- It fetches instructions from the memory.
- It transfers data to and from the memory and I/O.
- Maintains the 6-byte pre-fetch instruction queue(supports pipelining).

BIU mainly contains the 4 Segment registers, the Instruction Pointer, a pre-fetch queue, and an Address Generation Circuit.

Instruction Pointer (IP):

- It is a 16-bit register. It holds offset of the next instructions in the Code Segment.
- IP is incremented after every instruction byte is fetched.

- IP gets a new value whenever a branch instruction occurs.
- CS is multiplied by 10H to give the 20-bit physical address of the Code Segment.
- The address of the next instruction is calculated by using the formula CS x 10H + IP.

Example:
```
CS = 4321H IP = 1000H
then CS x 10H = 43210H + offset =  44210H
Here Offset = Instruction Pointer(IP)
```
This is the address of the next instruction.

Code Segment register: (16 Bit register): CS holds the base address for the Code Segment. All programs are stored in the Code Segment and accessed via the IP.

Data Segment register: (16 Bit register): DS holds the base address for the Data Segment.

Stack Segment register: (16 Bit register): SS holds the base address for the Stack Segment.

Extra Segment register: (16 Bit register): ES holds the base address for the Extra Segment.

Please note that segments are present in memory and segment registers are present in Microprocessor. Segment registers store starting address of each segments in memory.

Address Generation Circuit:
- The BIU has a Physical Address Generation Circuit.
- It generates the 20-bit physical address using Segment and Offset addresses using the formula:
- In Bus Interface Unit (BIU) the circuit shown by the Σ symbol is responsible for the calculation unit which is used to calculate the physical address of an instruction in memory.

Physical Address = Segment Address x 10H + Offset Address

6 Byte Pre-fetch Queue:
- It is a 6-byte queue (FIFO).
- Fetching the next instruction (by BIU from CS) while executing the current instruction is called pipelining.
- Gets flushed whenever a branch instruction occurs.
- The pre-Fetch queue is of 6-Bytes only because the maximum size of instruction that can have in 8086 is 6 bytes. Hence to cover up all operands and data fields of maximum size instruction in 8086 Microprocessor there is a Pre-Fetch queue is 6 Bytes.
- The pre-Fetch queue is connected with the control unit which is responsible for decoding op-code and operands and telling the execution unit what to do with the help of timing and control signals.
- The pre-Fetch queue is responsible for pipelining and because of that 8086 microprocessor is called fetch, decode, execute type microprocessor. Since there are always instructions present for decoding and execution in this queue the speed of execution in the microprocessor is gradually increased.
- When there is a 2-byte space in the instruction pre-fetch queue then only the next instruction will be pushed into the queue otherwise if only a 1-byte space is vacant then there will not be any allocation in the queue. It will wait for a spacing of 2 bytes in subsequent queue decoding operations.
- Instruction pre-fetch queue works in a sequential manner so if there is any branch condition

then in that situation pre-fetch queue fails. Hence to avoid chaos instruction queue is flushed out when any branch or conditional jumps occur.

prefetch unit:
The Prefetch Unit in the 8086 microprocessor is a component responsible for fetching instructions from memory and storing them in a queue. The prefetch unit allows the 8086 to perform multiple instruction fetches in parallel, improving the overall performance of the microprocessor.

The prefetch unit consists of a buffer and a program counter that are used to fetch instructions from memory. The buffer stores the instructions that have been fetched and the program counter keeps track of the memory location of the next instruction to be fetched. The prefetch unit fetches several instructions ahead of the current instruction, allowing the 8086 to execute instructions from the buffer rather than from memory.

This parallel processing of instruction fetches helps to reduce the wait time for memory access, as the 8086 can continue to execute instructions from the buffer while it waits for memory access to complete. This results in improved overall performance, as the 8086 is able to execute more instructions in a given amount of time.

The prefetch unit is an important component of the 8086 microprocessor, as it allows the microprocessor to work more efficiently and perform more instructions in a given amount of time. This improved performance helps to ensure that the 8086 remains competitive in its performance and capabilities, even as technology continues to advance.

The Execution Unit (EU):
The main components of the EU are General purpose registers, the ALU, Special purpose registers, the Instruction Register and Instruction Decoder, and the Flag/Status Register.

- Fetches instructions from the Queue in BIU, decodes, and executes arithmetic and logic operations using the ALU.
- Sends control signals for internal data transfer operations within the microprocessor.(Control Unit)
- Sends request signals to the BIU to access the external module.
- It operates with respect to T-states (clock cycles) and not machine cycles.

8086 has four 16-bit general purpose registers AX, BX, CX, and DX which store intermediate values during execution. Each of these has two 8-bit parts (higher and lower).

- AX register: (Combination of AL and AH Registers) It holds operands and results during multiplication and division operations. Also an accumulator during String operations.
- BX register: (Combination of BL and BH Registers) It holds the memory address (offset address) in indirect addressing modes.
- CX register: (Combination of CL and CH Registers) It holds the count for instructions like a loop, rotates, shifts and string operations.
- DX register: (Combination of DL and DH Registers) It is used with AX to hold 32-bit values during multiplication and division.

Arithmetic Logic Unit (16-bit): Performs 8 and 16-bit arithmetic and logic operations.

Special purpose registers (16-bit): Special purpose registers are called Offset registers also. Which points to specific memory locations under each segment.

We can understand the concept of segments as Textbook pages. Suppose there are 10 chapters in one textbook and each chapter takes exactly 100 pages. So the book will contain 1000 pages. Now suppose we want to access page number 575 from the book then 500 will be the segment base address which can be anything in the context of microprocessors like Code, Data, Stack, and Extra Segment. So 500 will be segment registers that are present in Bus Interface Unit (BIU). And 500 + 75 is called an offset register through which we can reach on specific page number under a specific segment.

Hence 500 is the segment base address and 75 is an offset address or (Instruction Pointer, Stack Pointer, Base Pointer, Source Index, Destination Index) any of the above according to their segment implementation.

- Stack Pointer: Points to Stack top. Stack is in Stack Segment, used during instructions like PUSH, POP, CALL, RET etc.
- Base Pointer: BP can hold the offset addresses of any location in the stack segment. It is used to access random locations of the stack.
- Source Index: It holds offset address in Data Segment during string operations.
- Destination Index: It holds offset address in Extra Segment during string operations.

Instruction Register and Instruction Decoder:
The EU fetches an opcode from the queue into the instruction register. The instruction decoder decodes it and sends the information to the control circuit for execution.

Flag/Status register (16 bits): It has 9 flags that help change or recognize the state of the microprocessor.

6 Status flags:

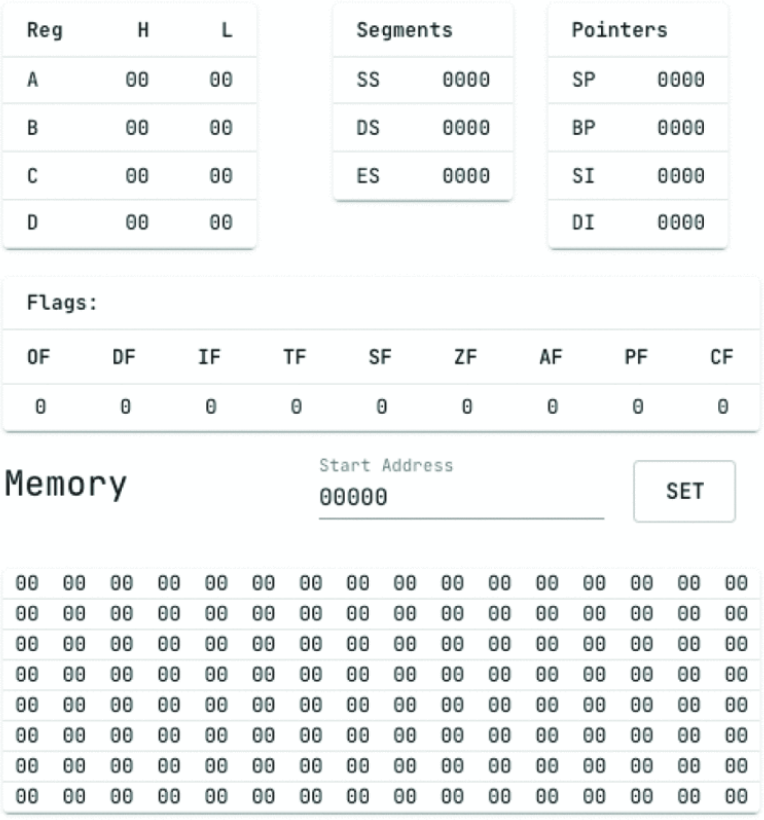
- Carry flag(CF)
- Parity flag(PF)
- Auxiliary carry flag(AF)
- Zero flag(Z)
- Sign flag(S)
- Overflow flag (O)

Status flags are updated after every arithmetic and logic operation.

Control flags:

- Trap flag(TF)
- Interrupt flag(IF)
- Direction flag(DF)

These flags can be set or reset using control instructions like CLC, STC, CLD, STD, CLI, STI, etc. The Control flags are used to control certain operations.



Intel 8086 registers

$^{1}_{9}$ $^{1}_{8}$ $^{1}_{7}$ $^{1}_{6}$ $^{1}_{5}$ $^{1}_{4}$ $^{1}_{3}$ $^{1}_{2}$ $^{1}_{1}$ $^{1}_{0}$ $^{0}_{9}$ $^{0}_{8}$ $^{0}_{7}$ $^{0}_{6}$ $^{0}_{5}$ $^{0}_{4}$ $^{0}_{3}$ $^{0}_{2}$ $^{0}_{1}$ $^{0}_{0}$ *(bit position)*			
Main registers			
	AH	AL	AX (primary accumulator)
0 0 0 0	BH	BL	**BX** (base, accumulator)
	CH	CL	**CX** (counter, accumulator)
	DH	DL	**DX** (accumulator, extended acc)
Index registers			
0 0 0 0	SI		**S**ource Index
0 0 0 0	DI		**D**estination Index
0 0 0 0	BP		**B**ase Pointer
0 0 0 0	SP		**S**tack Pointer
Program counter			
0 0 0 0	IP		**I**nstruction Pointer
Segment registers			
CS		0 0 0 0	**C**ode Segment
DS		0 0 0 0	**D**ata Segment
ES		0 0 0 0	**E**xtra Segment
SS		0 0 0 0	**S**tack Segment
Status register			
- - - - O D I T S Z - A - P - C			Flags

Decode unit:
The Decode Unit in the 8086 microprocessor is a component that decodes the instructions that have been fetched from memory. The decode unit takes the machine code instructions and translates them into micro-operations that can be executed by the microprocessor's execution unit.

The Decode Unit works in parallel with the Prefetch Unit, which fetches instructions from memory and stores them in a queue. The Decode Unit reads the instructions from the queue and translates them into micro-operations that can be executed by the microprocessor.

The Decode Unit is an important component of the 8086 microprocessor, as it allows the microprocessor to execute instructions efficiently and accurately. The decode unit ensures that the microprocessor can execute complex instructions, such as jump instructions and loop instructions, by translating them into a series of simple micro-operations.

The Decode Unit is responsible for decoding instructions, performing register-to-register operations, and performing memory-to-register operations. It also decodes conditional jumps, calls, and returns, and performs data transfers between memory and registers.

The Decode Unit helps to improve the performance of the 8086 microprocessor by allowing it to execute instructions quickly and accurately. This improved performance helps to ensure that the 8086 remains competitive in its performance and capabilities, even as technology continues to advance.

5.control unit :
The Control Unit in the 8086 microprocessor is a component that manages the overall operation of the microprocessor. The control unit is responsible for controlling the flow of instructions through the

microprocessor and coordinating the activities of the other components, including the Decode Unit, Execution Unit, and Prefetch Unit.

The Control Unit acts as the central coordinator for the microprocessor, directing the flow of data and instructions and ensuring that the microprocessor operates correctly. It also monitors the state of the microprocessor, ensuring that the correct sequence of operations is followed.

The Control Unit is responsible for fetching instructions from memory, decoding them, executing them, and updating the microprocessor's state. It also handles interrupt requests and performs system management tasks, such as power management and error handling.

The Control Unit is an essential component of the 8086 microprocessor, as it allows the microprocessor to operate efficiently and accurately. The control unit ensures that the microprocessor can execute complex instructions, such as jump instructions and loop instructions, by coordinating the activities of the other components.

The Control Unit helps to improve the performance of the 8086 microprocessor by managing the flow of instructions and data through the microprocessor, ensuring that the microprocessor operates correctly and efficiently. This improved performance helps to ensure that the 8086 remains competitive in its performance and capabilities, even as technology continues to advance.

The 8086 microprocessor uses three different buses to transfer data and instructions between the microprocessor and other components in a computer system. These buses are:

- Address Bus: The address bus is used to send the memory address of the instruction or data being read or written. The address bus is 16 bits wide, allowing the 8086 to address up to 64 kilobytes of memory.

- Data Bus: The data bus is used to transfer data between the microprocessor and memory. The data bus is 16 bits wide, allowing the 8086 to transfer 16-bit data words at a time.

- Control Bus: The control bus is used to transfer control signals between the microprocessor and other components in the computer system. The control bus is used to send signals such as read, write, and interrupt requests, and to transfer status information between the microprocessor and other components.

The buses in the 8086 microprocessor play a crucial role in allowing the microprocessor to access and transfer data from memory, as well as to interact with other components in the computer system. The 8086's ability to use these buses efficiently and effectively helps to ensure that it remains competitive in its performance and capabilities, even as technology continues to advance.

Execution of whole 8086 Architecture:

- All instructions are stored in memory hence to fetch any instruction first task is to obtain the Physical address of the instruction is to be fetched. Hence this task is done by Bus Interface Unit (BIU) and by Segment Registers. Suppose the Code segment has a Segment address and the Instruction pointer has some offset address then the physical address calculator circuit calculates the physical address in which our instruction is to be fetched.
- After the address calculation instruction is fetched from memory and it passes through C-Bus (Data bus) as shown in the figure, and according to the size of the instruction, the instruction pre-fetch queue fills up. For example MOV AX, BX is 1 Byte instruction so it will take only the 1st block of the queue, and MOV BX,4050H is 3 Byte instruction so it will take 3 blocks of the pre-

fetch queue.

- When our instruction is ready for execution, according to the FIFO property of the queue instruction comes into the control system or control circuit which resides in the Execution unit. Here instruction decoding takes place. The decoding control system generates an opcode that tells the microprocessor unit which operation is to be performed. So the control system sends signals all over the microprocessor about what to perform and what to extract from General and Special Purpose Registers.
- Hence after decoding microprocessor fetches data from GPR and according to instructions like ADD, SUB, MUL, and DIV data residing in GPRs are fetched and put as ALU's input. and after that addition, multiplication, division, or subtraction whichever calculation is to be carried out.
- According to arithmetic, flag register values change dynamically.
- While Instruction was decoding and executing from step-3 of our algorithm, the Bus interface Unit doesn't remain idle. it continuously fetches an instruction from memory and put it in a pre-fetch queue and gets ready for execution in a FIFO manner whenever the time arrives.
- So in this way, unlike the 8085 microprocessor, here the fetch, decode, and execution process happens in parallel and not sequentially. This is called pipelining, and because of the instruction pre-fetch queue, all fetching, decoding, and execution process happen side-by-side. Hence there is partitioning in 8086 architecture like Bus Interface Unit and Execution Unit to support Pipelining phenomena.

Advantages of Architecture of 8086:
The architecture of the 8086 microprocessor provides several advantages, including:

- Wide range of instructions: The 8086 microprocessor supports a wide range of instructions, allowing programmers to write complex programs that can perform many different operations.
- Segmented memory architecture: The segmented memory architecture allows the 8086 microprocessor to address large amounts of memory, up to 1 MB, while still using a 16-bit data bus.
- Powerful instruction set: The instruction set of the 8086 microprocessor includes many powerful instructions that can perform multiple operations in a single instruction, reducing the number of instructions needed to perform a given task.
- Multiple execution units: The 8086 microprocessor has two main execution units, the execution unit and the bus interface unit, which work together to efficiently execute instructions and manage data transfer.
- Rich set of registers: The 8086 microprocessor has a rich set of registers, including general-purpose registers, segment registers, and special registers, allowing programmers to efficiently manipulate data and control program flow.
- Backward compatibility: The architecture of the 8086 microprocessor is backward compatible with earlier 8-bit microprocessors, allowing programs written for these earlier microprocessors to be easily ported to the 8086 microprocessor.

Dis-advantages of Architecture of 8086:
The architecture of the 8086 microprocessor has some disadvantages, including:

- Complex programming: The architecture of the 8086 microprocessor is complex and can be difficult to program, especially for novice programmers who may not be familiar with the assembly language programming required for the 8086 microprocessor.
- Segmented memory architecture: While the segmented memory architecture allows the 8086 microprocessor to address a large amount of memory, it can be difficult to program and manage, as it requires programmers to use both segment registers and offsets to address memory.
- Limited performance: The 8086 microprocessor has a limited performance compared to modern

microprocessors, as it has a slower clock speed and a limited number of execution units.
- Limited instruction set: While the 8086 microprocessor has a wide range of instructions, it has a limited instruction set compared to modern microprocessors, which can limit its functionality and performance in certain applications.
- Limited memory addressing: The 8086 microprocessor can only address up to 1 MB of memory, which can be limiting in applications that require large amounts of memory.
- Lack of built-in features: The 8086 microprocessor lacks some built-in features that are commonly found in modern microprocessors, such as hardware floating-point support and virtual memory management.

Materials Used

- Emu8086 or similar 8086 emulator
- Intel 8086 instruction set reference
- ASCII table for character visualization
- PC Assembly Language – Paul A. Carter

Procedure

- The INT 10h BIOS interrupt provides low-level screen manipulation services.
- Function AH = 13h (Teletype output) writes a string of characters at a specified location.
- The message to be printed is stored in the data segment and accessed via the ES:BP pair.
- Registers used: CX (Length of the string), ES (Segment), BP (Offset), DL (Column position).
- Demonstrates BIOS-level control over text output.

- Define the string 'Hello World' using DB.
- Set AH = 13h for BIOS teletype output.
- Set CX = 11 for the string length.
- Set ES = 0 and BP = offset of the message.
- Set DL = 0 to start from the first column.
- Call INT 10h to print the message.

Simulation

THE CODE:

```
; Hello World program using BIOS interrupt
hello: DB "Hello World"

start:
MOV AH, 0x13
MOV CX, 11
MOV BX, 0
MOV ES, BX
MOV BP, OFFSET hello
MOV DL, 0
int 0x10
```

Expected Output:
A message "Hello World" will be displayed on the emulator screen using BIOS-level instructions.

Evaluation of Results

Answer the questions below after running the simulation.

- What is the role of INT 10h in this program?

- Why do we move BX = 0 before setting ES?
- What would happen if CX was set incorrectly?
- How does BIOS interrupt differ from DOS interrupt (INT 21h)?

Safety Precautions

References

- https://www.geeksforgeeks.org/architecture-of-8086/
- https://www.philadelphia.edu.jo/academics/qhamarsheh/uploads/emu8086.pdf
- https://ieeexplore.ieee.org/document/9824078
- https://en.wikipedia.org/wiki/Intel_8086#
- https://www.geeksforgeeks.org/computer-organization-architecture/different-instruction-cycles/
- https://www.geeksforgeeks.org/electronics-engineering/introduction-of-microprocessor/

2. Experiment 2 in 8086 Emulator: Performing Word-Length Addition in 8086 Assembly

Objective of the Experiment

To develop an assembly program in Emu8086 that adds two word-length constants and stores the result in memory. This experiment introduces register operations, arithmetic instructions, and memory addressing using the 8086 architecture.

- Work with word-length constants using DW
- Use 8086 registers for arithmetic (AX, BX)
- Store computed results in memory with MOV [DI], AX
- Visualize and verify outputs using emulator tools

Theoretical Background

The following 8086 assembly source code is for a subroutine named _strtolower that copies a null-terminated ASCIIZ character string from one location to another, converting all alphabetic characters to lower case. The string is copied one byte (8-bit character) at a time.

```
                        ; _strtolower:
                        ; Copy a null-terminated ASCII string, converting
                        ; all alphabetic characters to lower case.
                        ; ES=DS
                        ; Entry stack parameters
                        ;   [SP+4] = src, Address of source string
                        ;   [SP+2] = dst, Address of target string
                        ;   [SP+0] = Return address
                        ;
0000                    _strtolower proc
0000  55                            push    bp              ;Set up the call frame
0001  89 E5                         mov     bp,sp
0003  56                            push    si
0004  57                            push    di
0005  8B 75 06                     mov     si,[bp+6]       ;Set si = src (+2 due to push bp)
0008  8B 7D 04                     mov     di,[bp+4]       ;Set di = dst
000B  FC                            cld                     ;string direction ascending

000C  AC                loop:       lodsb                   ;Load al from [si], inc si
000D  3C 41                         cmp     al,'A'          ;If al < 'A',
000F  7C 06                         jl      copy            ; skip conversion
0011  3C 5A                         cmp     al,'Z'          ;If al > 'Z',
0013  7F 02                         jg      copy            ; skip conversion
0015  04 20                         add     al,'a'-'A'      ;Convert al to lowercase
0017  AA                copy:       stosb                   ;Store al to es:[di], inc di
0018  08 C0                         or      al,al           ;If al <> 0,
001A  75 F0                         jne     loop            ; repeat the loop

001C  5F                done:       pop     di              ;restore di and si
001D  5E                            pop     si
001E  5D                            pop     bp              ;Restore the prev call frame
001F  C3                            ret                     ;Return to caller
001F                                end     proc
```

The example code uses the BP (base pointer) register to establish a call frame, an area on the stack that contains all of the parameters and local variables for the execution of the subroutine. This kind of calling convention supports reentrant and recursive code and has been used by Algol-like languages since the late 1950s. A flat memory model is assumed, specifically, that the DS and ES segments address the same region of memory.

- Word-length data (16 bits) can be stored using the DW directive.
- Arithmetic operations are performed using CPU registers and the ADD instruction.
- Carry flag (CF) is cleared before addition to avoid previous overflow errors.
- The result is stored using memory addressing (MOV [DI], AX).
- Use of print reg to display register contents.

Materials Used

- Emu8086 emulator
- 8086 Instruction Set Summary
- Basic knowledge of hexadecimal representation and CPU registers

Procedure

- Define two word-length operands using DW.
- Load operands into AX and BX using MOV instructions.
- Clear the carry flag using CLC.
- Use ADD AX, BX to perform the addition.
- Store the result in memory using MOV [DI], AX.
- Print the result using print reg to verify the output.

Simulation

THE CODE:

```
; Program to add two word length numbers
OPR1: DW 0x6969
OPR2: DW 0x0420
RESULT: DW 0

start:
MOV AX, word OPR1
MOV BX, word OPR2
CLC
ADD AX, BX
MOV DI, OFFSET RESULT
MOV word [DI], AX
print reg
```

Expected Output:

The result of the addition 0x6969 + 0x0420 = 0x6D89 will be stored in RESULT and displayed in the AX register.

Evaluation of Results

- Why is the carry flag cleared before addition?
- What is the role of the OFFSET keyword in storing results?
- How would the result differ if operands were byte-sized?
- How is memory access performed with segment and offset?
- What are the advantages of using registers for arithmetic?

Safety Precautions

References

- https://www.geeksforgeeks.org/architecture-of-8086/
- https://www.philadelphia.edu.jo/academics/qhamarsheh/uploads/emu8086.pdf
- https://ieeexplore.ieee.org/document/9824078
- https://en.wikipedia.org/wiki/Intel_8086#

3. Experiment 3 in 8086 Emulator: Transferring Data Between Segments in 8086 Assembly Language

Objective of the Experiment

To write and execute an assembly program that transfers a block of data from one memory segment to another using segment registers (DS, ES) and offset registers (SI, DI). The experiment demonstrates register-based memory addressing, loop control, and basic data movement.
In this lab, you have learned how to:

- Use segment and offset registers for memory access
- Create a loop to copy bytes from one memory area to another
- Understand segment-to-segment data movement in 8086
- Visualize the effect of data manipulation using memory inspection

This experiment reinforces key concepts in low-level memory management and instruction flow control in x86 architecture.

Theoretical Background

In the 8086 architectures, memory is accessed using segment:offset addressing. Each segment (data, extra, stack, etc.) is pointed to by specific segment registers (DS, ES, etc.). To copy data from one segment to another:

- DS:SI is used to point to the source
- ES:DI is used to point to the destination

The MOV, INC, DEC, and JNZ instructions are used to create a loop that moves each byte one at a time.

Materials Used

- 8086 emulator (e.g., yjdoc2.github.io/8086-emulator-web or PCjs.org)

Procedure

- Initialize Segments and Define Data

```
SET 0
src: DB 0x3
   DB 0x5
   DB 0x7
SET 0x1
dest: DB [0,3]  ; Reserve 3 bytes for destination
```

- Print Initial Memory State

```
PRINT MEM 0:8
PRINT MEM 0x10:8
```

- Set Up Registers

```
MOV AX, 0
MOV DS, AX
MOV AX, 0x1
MOV ES, AX
MOV SI, OFFSET src
MOV DI, OFFSET dest
MOV CX, 0x3
```

- Data Transfer Loop

```
_loop:
MOV AH, BYTE DS[SI]
MOV BYTE ES[DI], AH
INC SI
```

```
INC DI
DEC CX
JNZ _loop
```

- Print Final Memory State

```
PRINT MEM 0:8
PRINT MEM 0x10:8
```

Simulation

Expected Output:

Initial Memory (Segment 0):		Initial Memory (Segment 1):		Final Memory (Segment 1):	
Address	**Value**	**Address**	**Value**	**Address**	**Value**
0000	0x03	0000	0x00	0000	0x03
0001	0x05	0001	0x00	0001	0x05
0002	0x07	0002	0x00	0002	0x07

Evaluation of Results

- What is the purpose of the SI and DI registers?
- How many times does the loop run? Why?
- What would happen if CX were set to 5 instead of 3?
- How does using ES differ from using DS only?
- What happens if DS and ES are set to the same segment?

Safety Precautions

References

- https://www.geeksforgeeks.org/architecture-of-8086/
- https://www.philadelphia.edu.jo/academics/qhamarsheh/uploads/emu8086.pdf
- https://ieeexplore.ieee.org/document/9824078

4. Experiment 4 in 8086 Emulator: Finding the Minimum and Maximum Values in a Data Set Using 8086 Assembly Language

Objective of the Experiment

To write and execute an 8086 Assembly program that iterates over a set of byte values stored in memory and determines both the minimum and maximum values. The experiment demonstrates comparison instructions, register-based iteration using loops, and memory storage of computed results.

Theoretical Background

8086 assembly allows direct access to memory and registers for comparison and data manipulation. Using CMP, JNC, and conditional branching (LOOP, JMP), we can scan through a list and find:

- The minimum value: by comparing and updating if a smaller value is found.
- The maximum value: by comparing and updating if a larger value is found.

Key registers used:

- SI: source index for iterating through the data array
- CX: loop counter
- AL: temporary register to store current min/max
- MIN, MAX: memory locations to store the final result

Materials Used

- 8086 emulator (e.g., yjdoc2.github.io, PCjs.org, Emu8086)

Procedure

- Data Definition

```
vals:
DB 0x12, 0x34, 0x78, 0x13, 0x99, 0x65, 0x85, 0x11, 0x84, 0x36
last: DB 0            ; used to mark end, and count total items
MIN:  DB 0            ; to store minimum value
MAX:  DB 0            ; to store maximum value
```

- Find Minimum Value

```
MOV SI, 0
MOV CX, OFFSET last
MOV AL, BYTE [SI]     ; Initial value for comparison
back:
CMP BYTE [SI], AL     ; Compare current value with AL (min)
JNC skip              ; If current >= AL, skip
MOV AL, BYTE [SI]     ; Update AL if current is smaller
skip:
INC SI
LOOP back
MOV BYTE MIN, AL      ; Store result in MIN
```

- Find Maximum Value

```
MOV SI, 0
MOV CX, OFFSET last
MOV AL, BYTE [SI]     ; Reset AL for max search
back1:
CMP AL, BYTE [SI]     ; Compare AL with current
JNC skip1             ; If AL >= current, skip
MOV AL, BYTE [SI]     ; Update AL if current is larger
skip1:
INC SI
```

```
LOOP back1
MOV BYTE MAX, AL      ; Store result in MAX
```
- Display Memory for Results

```
PRINT MEM :15     ; Assuming emulator supports memory print
```

Simulation

Initial Data in Memory:

[0x12, 0x34, 0x78, 0x13, 0x99, 0x65, 0x85, 0x11, 0x84, 0x36]

- MIN → 0x11 (17 in decimal)
- MAX → 0x99 (153 in decimal)

Memory at MIN and MAX labels should reflect these values after execution.

Evaluation of Results

This experiment demonstrates how loops and conditional instructions in 8086 Assembly can be used to traverse and evaluate arrays. It emphasizes practical memory addressing (SI, OFFSET), comparison (CMP), and conditional execution (JNC) to derive meaningful results from raw data. Therefore, answer the following questions.

- How does JNC help in finding minimum values?
- What does OFFSET last provide in this context?
- Why is AL reset before finding the maximum?
- What would happen if CX were incorrect (e.g., too large)?
- How could this program be extended to find the average?

Safety Precautions

References

- https://www.geeksforgeeks.org/architecture-of-8086/
- https://www.philadelphia.edu.jo/academics/qhamarsheh/uploads/emu8086.pdf
- https://ieeexplore.ieee.org/document/9824078

5. Experiment 1 in Proteus: LED Blinking Simulation

Objective of the Experiment

To simulate the blinking of an LED using the Intel 8086 microprocessor and the 8255 Programmable Peripheral Interface (PPI) within the Proteus environment. The program is written in assembly language using Emu8086.

Theoretical Background

The 8086 microprocessor communicates with peripherals using shared address/data buses. It lacks built-in I/O ports, so peripheral chips like the 8255 PPI are used to provide programmable I/O lines.
To separate the address and data lines (which are multiplexed on AD0–AD7), a latch like the 74HC373 is used with the ALE (Address Latch Enable) signal.

The 8255 PPI provides three ports:

- PORTA, PORTB, PORTC, each 8-bit wide.
- It can operate in Mode 0 (basic input/output), Mode 1 (strobed), or Mode 2 (bidirectional).

In this experiment:

- PORTA is set as output.
- A single LED is connected to PA0.
- The microprocessor writes 00H and FFH alternately to PORTA to toggle the LED.

Materials Used

- Intel 8086 (U1)
- 74HC373 Latch (U2)
- 8255 PPI (U3)
- LED (D1)
- Proteus Design Suite
- Emu8086 Assembler

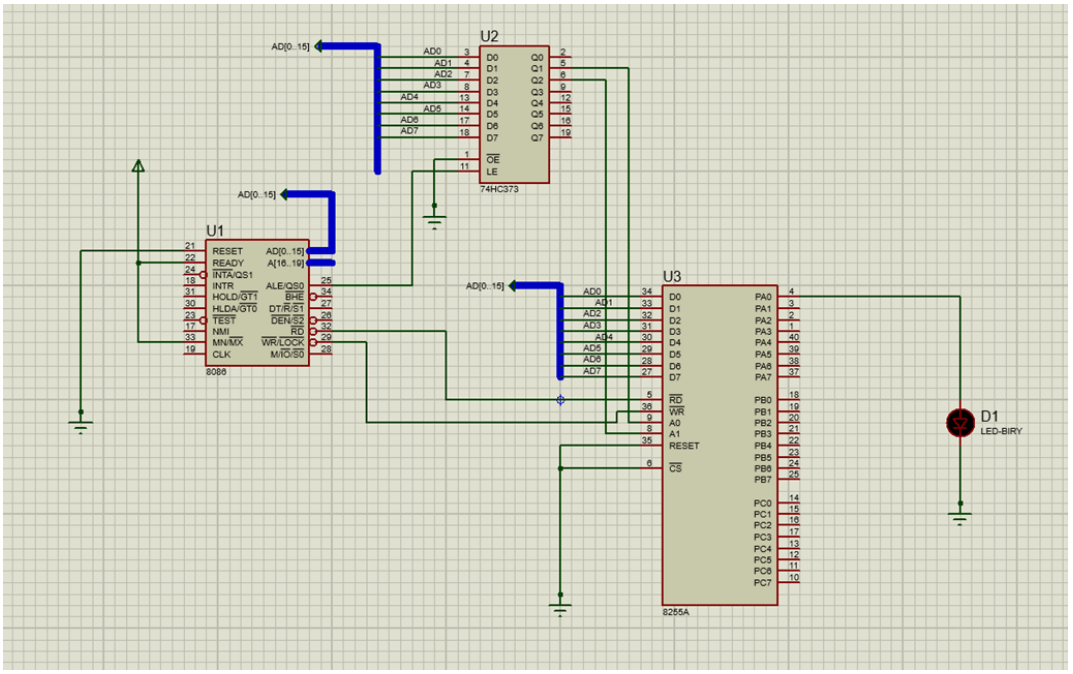


Figure 1. Pin connection diagram

Procedure

Circuit Construction in Proteus

Connect the 8086 microprocessor.
Add the 74HC373 to latch the lower address byte from AD0–AD7 using the ALE signal.
Connect the 8255 PPI:
Data bus: AD0–AD7
Control signals: WR, RD, RESET, CS, A0, A1
PORTA (PA0) to LED
4Assign proper addressing to PORTA at 00H and control register at 06H.

Assembly Code in Emu8086

```
;Ledblinking program
DATA SEGMENT
      PORTA EQU 00H
      PORTB EQU 02H
      PORTC EQU 04H
      PORT_CON EQU 06H
DATA ENDS
CODE SEGMENT
      MOV AX,DATA
      MOV DS,AX

ORG 000H
START:
      MOV DX,PORT_CON
      MOV AL, 10000000B
      OUT DX,AL

      JMP XX
XX:
      MOV AL,000H
      MOV DX,PORTA
      OUT DX,AL
      MOV CX,0DF36H
LOOPY1:LOOP LOOPY1
      MOV AL,OOFFH
      MOV DX,PORTA
      OUT DX,AL
      MOV CX,0DF36H
LOOPY2:LOOP LOOPY2
JMP XX

CODE ENDS
END
```

Simulation

Compile and create .bin file in Emu8086.
Load the file into the Proteus 8086 simulation environment.
Run the simulation.
Observe the LED blinking connected to PA0.

Evaluation of Results

The LED on PORTA.0 (PA0) blinks correctly as the output is toggled.
The 8255 responds to control word 10000000B correctly.
The delay loop introduces visible toggling.
The use of memory-mapped I/O is verified via correct port selection.

Safety Precautions

References

- Intel 8086 Datasheet
- Intel 8255A Datasheet
- PC Assembly Language - Paul A. Carter
- Emu8086 Documentation
- Proteus Simulation Manual
- https://youtube.com/playlist?list=PL-GE-cpvo9ywZ9BAA6F8DhCwMMjx_2uV0&si=t72Otboed1hHZefa
- https://github.com/osemrt/8086-Microprocessor

6. Experiment 2 in Proteus: Stepper Motor Control via 825

Objective of the Experiment

The aim of this experiment is to simulate the control of a unipolar stepper motor using the Intel 8086 microprocessor and 8255A Programmable Peripheral Interface (PPI) in the Proteus environment. The control logic is written in Assembly language using the Emu8086 emulator.

Theoretical Background

Stepper motors are digital actuators that move in discrete angular steps in response to input signals. Each step corresponds to a unique bit pattern applied to the motor's windings. This makes them ideal for precise position control applications.

Since the Intel 8086 does not have built-in I/O ports, the 8255A PPI is used to provide programmable I/O. The 8255 has three 8-bit ports: PORTA, PORTB, and PORTC, which can be programmed as input or output in various modes.

To separate the multiplexed address/data bus (AD0–AD7) of the 8086, a 74HC373 latch is used. This latch captures the lower byte of the address during the ALE (Address Latch Enable) signal and holds it stable for peripheral addressing.

In this experiment:

- PORTA (PA0–PA3) is used to send control signals to the stepper motor.
- The control pattern cycles through 8 positions (0° to 360°) to create full rotation.
- Motor control is observed through a LogicState display and the simulated stepper motor movement.

Materials Used

Intel 8086 Microprocessor (U1)
74HC373 Latch (U2)
8255A Programmable Peripheral Interface (U3)
Unipolar Stepper Motor
LogicState Probe
Proteus Design Suite
Emu8086 Assembler

Procedure

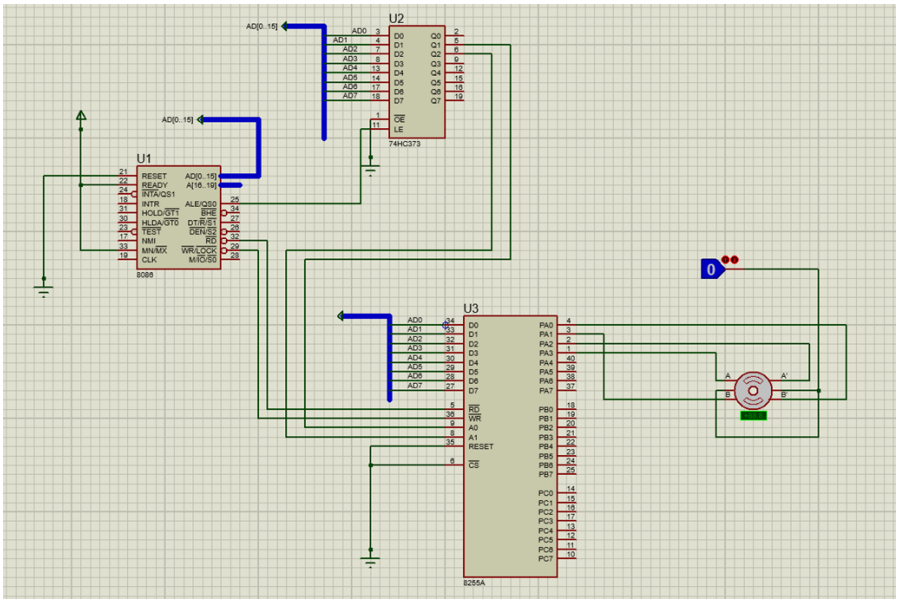


Figure 2. Pin connection diagram

Circuit Setup in Proteus

Add the 8086 microprocessor (U1) and connect its AD0–AD7 lines to both the 74HC373 and the 8255.

Connect ALE (pin 25) from 8086 to LE (pin 11) of the 74HC373 latch (U2).
Connect the Q0–Q7 outputs of the latch to the A0, A1, and CS pins of the 8255 (U3).
Connect RD (pin 28) and WR (pin 29) from 8086 to the appropriate control pins of 8255.
Connect PORTA's PA0–PA3 to the stepper motor's control inputs (A, A', B, B').
Add a LogicState probe to observe the bit pattern changes sent to the motor.
Ensure CS, RESET, and GND lines are properly tied.

Assembly Code Using Emu8086

```
; Stepper Motor Control via 8086 and
8255

DATA SEGMENT
 PORTA     EQU 00H
 PORTB     EQU 02H
 PORTC     EQU 04H
 PORT_CON  EQU 06H
DATA ENDS

CODE SEGMENT
 MOV AX, DATA
 MOV DS, AX

ORG 0000H
START:
 MOV DX, PORT_CON
 MOV AL, 10000000B
 OUT DX, AL

 JMP MAIN_LOOP

MAIN_LOOP:
 ; Step 1 - 0°
 MOV AL, 03H
 MOV DX, PORTA
 OUT DX, AL
 MOV CX, 0DF36H
Delay0: LOOP Delay0

 ; Step 2 - 45°
 MOV AL, 0BH
 OUT DX, AL
 MOV CX, 0DF36H
Delay1: LOOP Delay1

 ; Step 3 - 90°
 MOV AL, 0AH
 OUT DX, AL
 MOV CX, 0DF36H
Delay2: LOOP Delay2

 ; Step 4 - 135°
 MOV AL, 0EH
 OUT DX, AL
 MOV CX, 0DF36H
Delay3: LOOP Delay3

 ; Step 5 - 180°
 MOV AL, 0CH
 OUT DX, AL
 MOV CX, 0DF36H
Delay4: LOOP Delay4

 ; Step 6 - 225°
 MOV AL, 0DH
 OUT DX, AL
 MOV CX, 0DF36H
Delay5: LOOP Delay5

 ; Step 7 - 270°
 MOV AL, 05H
 OUT DX, AL
 MOV CX, 0DF36H
Delay6: LOOP Delay6

 ; Step 8 - 315°
 MOV AL, 07H
 OUT DX, AL
 MOV CX, 0DF36H
Delay7: LOOP Delay7

 ; Loop back to 0°
 JMP MAIN_LOOP

CODE ENDS
END
```

Simulation Steps

Compile the code in Emu8086 and export the .bin file.
Load the file into the 8086 microprocessor in Proteus.
Run the simulation.
Observe the stepper motor rotating in full steps, and the LogicState probe indicating the step sequence.

Evaluation of Results

The stepper motor rotates in full 45° increments based on the bit pattern provided to its coils.
The output values (03H, 0BH, 0AH, etc.) match the excitation sequence for full stepping.
Delay loops provide visible motor movement and prevent skipped steps.
Communication between the 8086 and 8255 is verified through successful motor control.

Safety Precautions

References

- Intel 8086 Microprocessor Datasheet
- Intel 8255A PPI Datasheet
- PC Assembly Language - Paul A. Carter
- Emu8086 User Guide
- Proteus Simulation Documentation
- https://youtube.com/playlist?list=PL-GE-cpvo9ywZ9BAA6F8DhCwMMjx_2uV0&si=t72Otboed1hHZefa
- https://github.com/osemrt/8086-Microprocessor

7. Experiment 3 in Proteus: Analog to Digital Converter with 8086 via 8255 PPI

Objective of the Experiment

The goal of this experiment is to interface the ADC0804 analog-to-digital converter with the Intel 8086 microprocessor using the 8255A Programmable Peripheral Interface. The digital output of the ADC is read via PORTA and displayed on PORTC.

Theoretical Background

The ADC0804 is an 8-bit analog-to-digital converter. It converts a continuous analog voltage into an 8-bit digital equivalent. The 8086 microprocessor does not have direct analog input capability; therefore, ADC interfacing is essential.

To handle I/O with the ADC0804, the 8255A PPI is used. It allows 8086 to control digital signals (RD, WR) and receive ADC output over the data bus. The ADC requires control signals (WR = LOW to start conversion, RD = LOW to read data) and a clock signal to function.

Interfacing Steps:

- Analog input is given to ADC0804 VIN+.
- Conversion is triggered by toggling WR from HIGH to LOW to HIGH.
- After conversion, the result is read via RD signal.

Materials Used

Intel 8086 (U4)
8255 PPI (U2)
74HC373 Latch (U3)
ADC0804 (U1)
Potentiometer (RV2)
Digital Voltmeter
Clock Source (for ADC)

Procedure

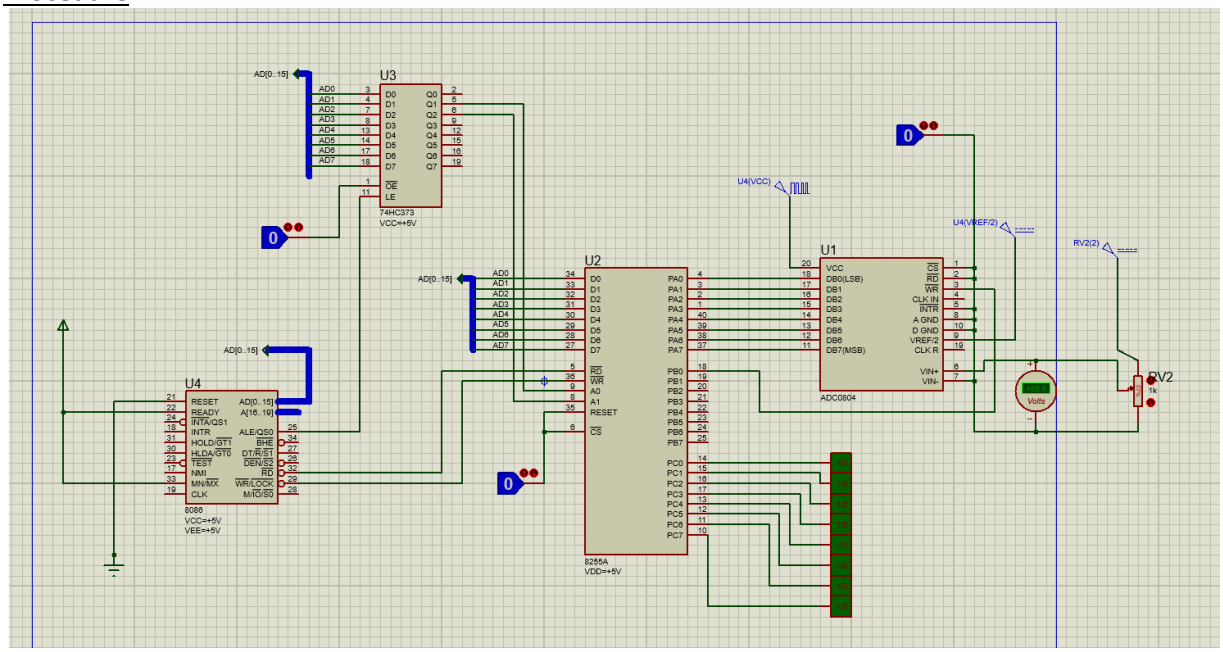


Figure 3 Pin connection diagram

Circuit Implementation in Proteus

The analog voltage is provided via a potentiometer connected to VIN+ of ADC0804.
VREF/2 pin is connected to 2.5V to set the reference for 0–5V conversion range.

CLK IN of ADC0804 receives a square wave clock (e.g., 640 kHz).
ADC output pins (DB0–DB7) are connected to PORTA of 8255.
PORTB is used to send control signals (RD, WR).
PORTC displays the converted 8-bit result.

Assembly Code (Emu8086)

```
DATA SEGMENT
 PORTA     EQU 00H
 PORTB     EQU 02H
 PORTC     EQU 04H
 PORT_CON  EQU 06H
DATA ENDS
CODE SEGMENT
 MOV AX, DATA
 MOV DS, AX

ORG 0000H

START:
 MOV DX, PORT_CON
 MOV AL, 10010000B
 OUT DX, AL

 MOV AL, 00H
```

```
XX:
 ; Read ADC0804 output to AL from
PORTA
 MOV DX, PORTA
 IN  AL, DX

 ; Output digital value to PORTC
 MOV DX, PORTC
 OUT DX, AL

 ; Trigger ADC WR = 0 → 1
 MOV DX, PORTB
 MOV AL, 00000000B    ; WR = 0, RD = 0
 OUT DX, AL

 MOV CX, 0FFH         ; Short delay
D1: LOOP D1

 MOV DX, PORTB
 MOV AL, 00000001B    ; WR = 1, RD = 0
 OUT DX, AL

 MOV CX, 0FFH         ; Short delay
D2: LOOP D2

 JMP XX               ; Repeat forever

CODE ENDS
END
```

Simulation Steps
Compile the above Assembly code using Emu8086.
Export the .bin file and load it into the 8086 CPU in Proteus.
Start the simulation.
Change RV2 to vary analog input.
Observe the digital voltage output (converted 8-bit) reflected on PORTC.

Evaluation of Results
When the analog voltage at VIN+ is changed via the potentiometer, the digital output on PORTC changes accordingly.
The voltmeter value and the binary pattern on PORTC match (e.g., 2.5V ≈ 128, 5V ≈ 255).
Control logic successfully triggers ADC conversion and reads the result.

Safety Precautions

References

- Intel ADC0804 Datasheet

- Intel 8255 PPI Datasheet
- Emu8086 Programming Manual
- PC Assembly Language - Paul A. Carter
- https://youtube.com/playlist?list=PL-GE-cpvo9ywZ9BAA6F8DhCwMMjx_2uV0&si=t72Otboed1hHZefa
- https://github.com/osemrt/8086-Microprocessor

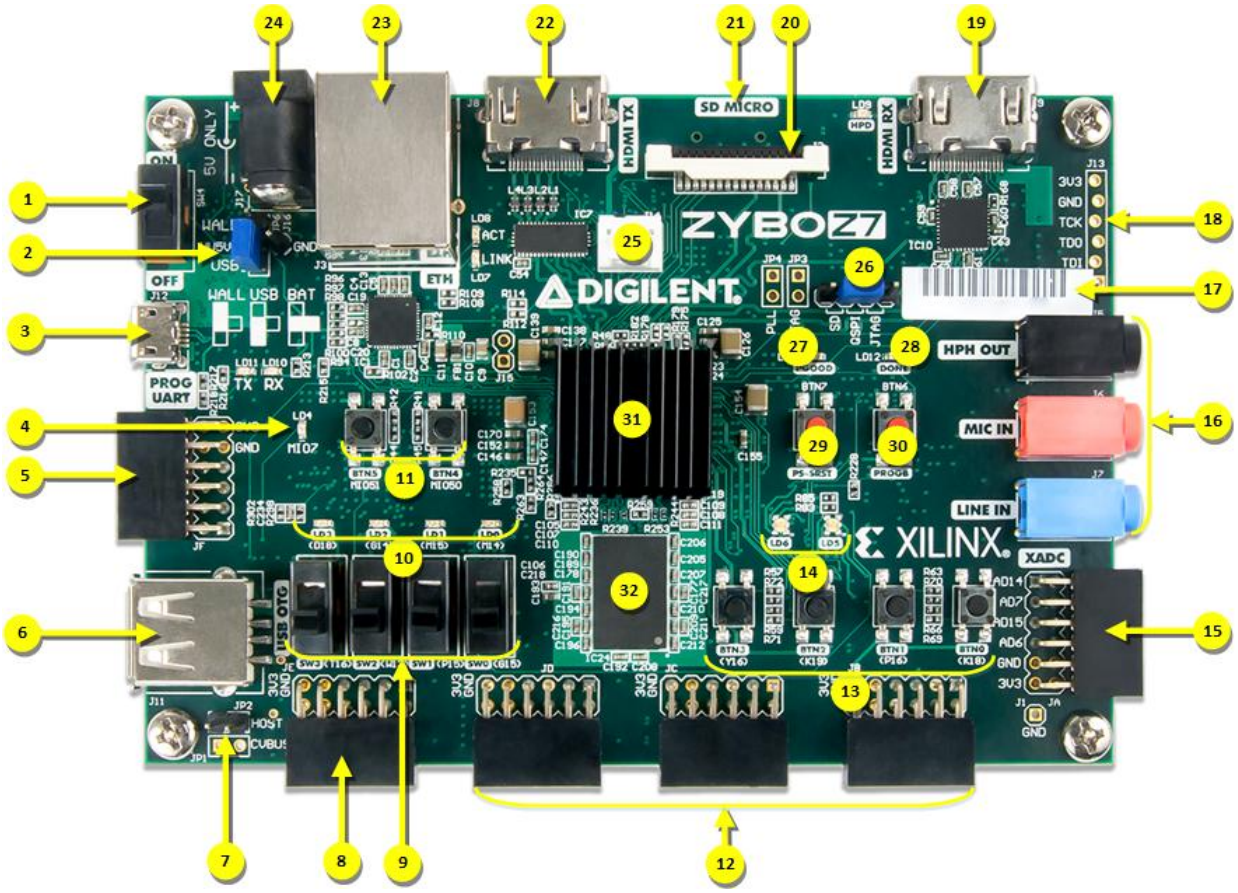
8. Experiment 1 on ZYBOZ7 Development Board: FPGA

Objective of the Experiment

This lab exercise aims to familiarize you with the Xilinx FPGA design flow via Vivado by stepping through a simple example. We will use Vivado to create hardware that lights up the LEDs on the ZYBO board depending on the status of the on-board DIP switches. The hardware will be using an FPGA and will be designed in Vivado using Verilog. After completing the aforementioned example, you will be expected to implement a simple counter and jackpot game on your own with the knowledge gained from the first part of this lab.

Theoretical Background

The Zybo Z7 is a feature-rich, ready-to-use embedded software and digital circuit development board built around the Xilinx Zynq-7000 family. The Zynq family is based on the Xilinx All Programmable System-on-Chip (AP SoC) architecture, which tightly integrates a dual-core ARM Cortex-A9 processor with Xilinx 7-series Field Programmable Gate Array (FPGA) logic. The Zybo Z7 surrounds the Zynq with a rich set of multimedia and connectivity peripherals to create a formidable single-board computer, even before considering the flexibility and power added by the FPGA. The Zybo Z7's video-capable feature set, including a MIPI CSI-2 compatible Pcam connector, HDMI input, HDMI output, and high DDR3L bandwidth, was chosen to make it an affordable solution for the high end embedded vision applications that Xilinx FPGAs are popular for. Attaching additional hardware is made easy by the Zybo Z7's Pmod connectors, allowing access to Digilent's catalog of over 70 Pmod peripheral boards, including motor controllers, sensors, displays, and more.

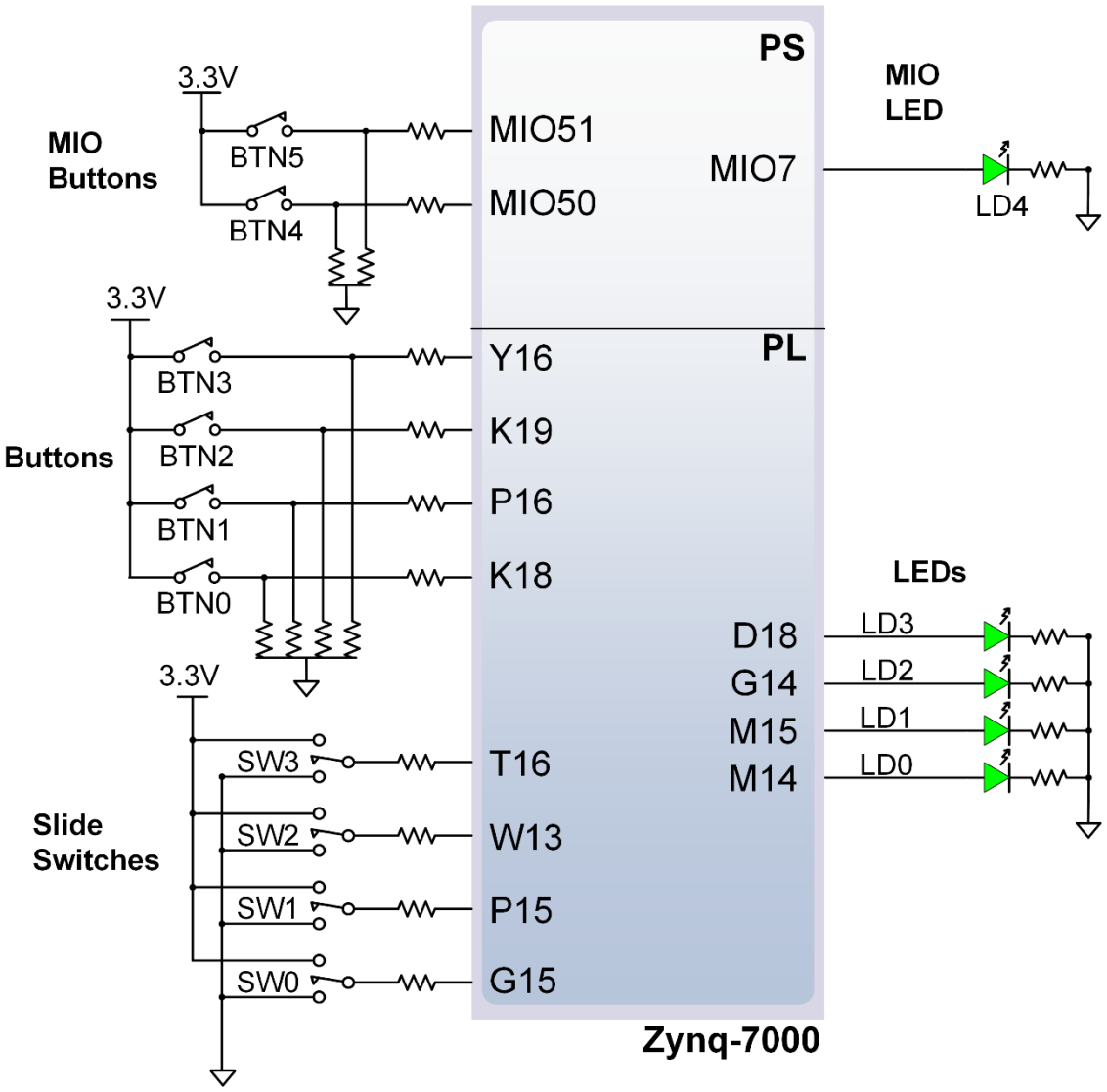


Callout	Description	Callout	Description	Callout	Description
1	Power Switch	12	High-speed Pmod ports *	23	Ethernet port
2	Power select jumper	13	User buttons	24	External power supply connector
3	USB JTAG/UART port	14	User RGB LEDs *	25	Fan connector (5V, three-wire) *
4	MIO User LED	15	XADC Pmod port	26	Programming mode select jumper
5	MIO Pmod port	16	Audio codec ports	27	Power supply good LED
6	USB 2.0 Host/OTG port	17	Unique MAC address label	28	FPGA programming done LED
7	USB Host power enable jumper	18	External JTAG port	29	Processor reset button
8	Standard Pmod port	19	HDMI input port	30	FPGA clear configuration button
9	User switches	20	Pcam MIPI CSI-2 port	31	Zynq-7000
10	User LEDs	21	microSD connector (other side)	32	DDR3L Memory
11	MIO User buttons	22	HDMI output port	** denotes difference between Z7-10 and Z7-20*	

The Zybo Z7 board includes four slide switches, four push-buttons, four individual LEDs, and two tri-color LEDs connected to the Zynq PL, as shown in Figure 13.1 (the Zybo Z7-10 only has one tri-color LED). There are also two pushbuttons and one LED connected directly to the PS via MIO pins, also shown in Figure below. The push-buttons and slide switches are connected to the Zynq via series resistors to prevent damage from inadvertent short circuits (a short circuit could occur if a pin assigned to a push-button or slide switch was inadvertently defined as an output). The push-buttons are “momentary” switches that normally generate a low output when they are at rest, and a high output only when they are pressed. Slide switches generate constant high or low inputs depending on their position.

The high-efficiency LEDs are anode-connected to the Zynq via 330-ohm resistors, so they will turn on when a logic high voltage is applied to their respective I/O pin. Additional LEDs that are not user-accessible indicate power-on, FPGA programming status, and USB and Ethernet port status.

The LED and two pushbuttons attached directly to the PS are accessed using the Zynq GPIO controller. This core is described in full in Chapter 14 of the Zynq Technical Reference Manual.



Materials Used

- Zynq-7000 ARM/FPGA SoC Development Board
- USB Programming cables, USB UART cables, and Power Supply, as required by the board.
- Vivado and Vitis, the development environments

Procedure

- Create a folder for your lab work.
- Launch the Vivado and create a new design project. Open a terminal window in the CentOS workstation and run the following commands:

  >source /softwares/Linux/xilinx/Vivado/2015.2/settings64.sh
  >Vivado
- The first sets up the environment in order to run Vivado and the second command starts the Vi vado Suite
- Once in Vivado, select File Create New Project. The New Project Wizard opens. Click Next. (Figure 1).

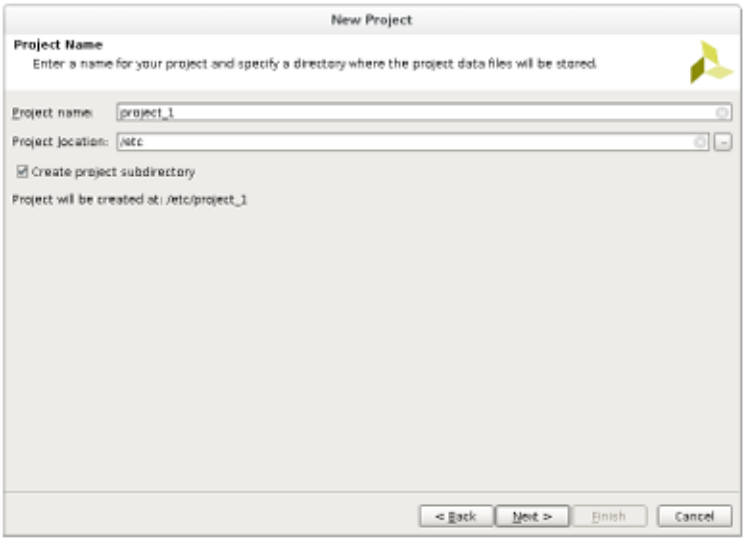


Figure 1: Create New Project

- Select a Project Name (ex. Lab8) and a Project Location (ex. /lab8 in your home directory). Then check the create project subdirectory and click Next. Select RTL project and leave 'Do not specify sources' unchecked at this time. Click Next. You will see 'Add Sources' window, select 'Target language' as Verilog and 'Simulator language' as Mixed. Click on the green '+' button and select 'Create file', a window pop up will appear. Select 'File type' as verilog, 'File name' as 'switch', and select 'File location' as '', click 'OK' to create the Verilog source file. Click 'Next' and you will see the 'Add Existing IP' window. Right now we don't need any IP. Click 'Next' and the 'Add Constraints(optional)' window will appear. We will add the Constraints File later in the lab. Click 'Next'.
- Next, the 'Default Part' window appears (Figure 2). We can select our hardware from the 'Parts' tab or from the 'Boards' tab. The 'Parts' tab lists Xilinx supported Parts(FPGAs) and the 'Boards' tab lists the supported boards. The ZYBO (Zynq Board) is an entry-level digital circuit development platform built around the Xilinx Zynq-7000 family, the Z-7010. The Z-7010 is based on the Xilinx All Programmable System-on-Chip (AP SoC) architecture, which integrates a dual-core ARM Cortex-A9 processor with a Xilinx 7-series Field Programmable Gate Array (FPGA) logic. In this lab and the next we will select the hardware from the 'Parts' tab and in the later labs we will select the hardware using the 'Boards' tab. Select the hardware using the following parameters.

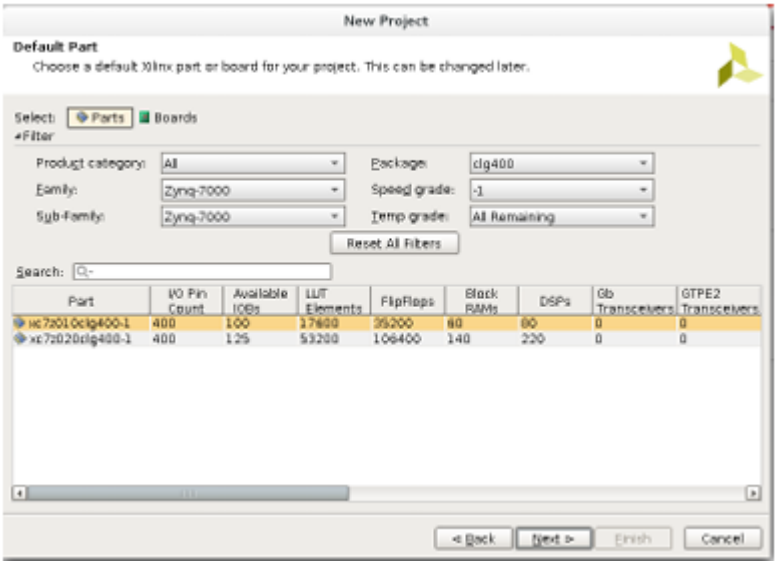


Figure 2: Device Properties

- Set the device properties to the following:
    - Device Family: Zynq-7000
    - Sub-Family: Zynq-7000
    - Package: clg400
    - Speed Grade:-1

- You will see two devices. Select the first device with part number ‘xc7z010clg400-1’ and click ‘Next’. Finally, review the information in the ‘New Project Summary’ window and hit ‘Finish’ to create project. Next, the ‘Define Module’ window appears (Figure 3). This allows us to define the ports for our hardware module. Xilinx will then auto generate part of our source file based on the information provided. Specify a port called ‘SWITCHES’. Set its direction to ‘input’, check Bus, set the Most Significant Bit (MSB) to 3, and set the Least Significant Bit (LSB) to 0. Specify another port called ‘LEDS’ and set the direction to ‘output’, check ‘Bus’, set MSB to 3, and set LSB to 0. This will create a 4-bit input port, which will connect to the on-board slide switches, and a 4-bit output port, which will connect to the on-board LEDs. Click ‘OK’

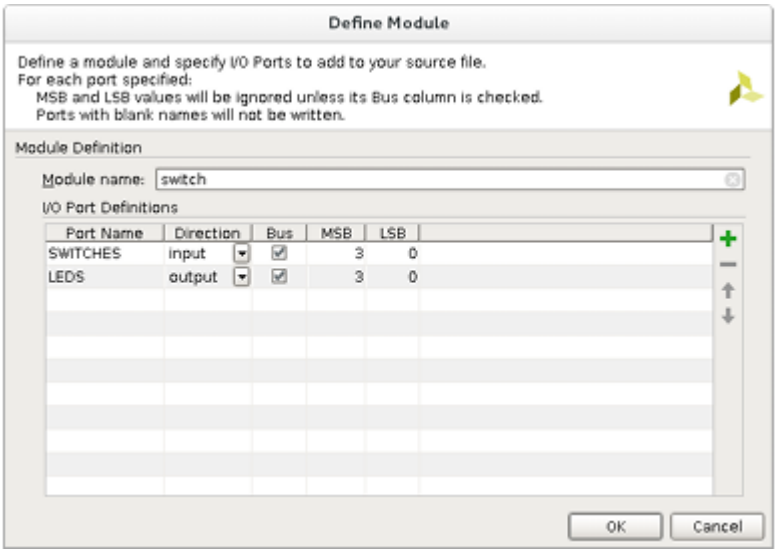


Figure 3: Define Module

Simulation

Vivado has created a new project and source file for us to modify. We will now create code to provide the desired functionality (i.e. to turn on LED[i] when Switch[i] is high).

- From the 'Sources' window, open the 'switch.v' file. It will contain a Verilog module with the port declarations described above.

- Above 'endmodule', add the following line of code: assign
  LEDS [3:0] = SWITCHES [3:0];

- The resulting verilog module should be as follows:

```
module switch (SWITCHES, LEDS);
        input [3:0] SWITCHES;
        output [3:0] LEDS;
        assign LEDS [3:0] = SWITCHES [3:0];
endmodule
```

- Click on File, then Save All files to save your changes. Saving the source file will perform a 'Syntax Check'. When you save the file, if you have made any syntax errors in the source file, Vivado will show error messages corresponding to syntax errors in the messages panel. Please clear the errors and save your source file by pressing Ctrl+S or Click on File, then Save All files.

- We now need to create the 'Xilinx Design Constraints(XDC)' file containing the location of the DIP switches and LEDs on the ZYBO Board. The .xdc file will be used to connect signals described in the Verilog file (LEDS[3:0] and SWITCHES[3:0] in our case) to pins on the FPGA, which are hardwired to the LEDS and DIP switches on the ZYBO board.

- Use your favorite text editor to create a new file called 'switch.xdc' in your lab8 project directory and copy the following text into the new file:

```
##Switches
set_property PACKAGE_PIN G15 [get_ports {SWITCHES[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SWITCHES[0]}]

set_property PACKAGE_PIN P15 [get_ports {SWITCHES[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SWITCHES[1]}]

set_property PACKAGE_PIN W13 [get_ports {SWITCHES[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SWITCHES[2]}]

set_property PACKAGE_PIN T16 [get_ports {SWITCHES[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SWITCHES[3]}]

##LEDs
set_property PACKAGE_PIN M14 [get_ports {LEDS[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDS[0]}]

set_property PACKAGE_PIN M15 [get_ports {LEDS[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDS[1]}]

set_property PACKAGE_PIN G14 [get_ports {LEDS[2]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {LEDS[2]}]

set_property PACKAGE_PIN D18 [get_ports {LEDS[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LEDS[3]}]
```

- After saving 'switch.xdc', return to the Sources panel in vivado, right click on the constraints folder and select 'add sources'. Next, the 'Add Sources ' window will open. Select 'Add or create constraints' and click 'Next'. Click on the green + button and select 'Add files' and navigate to the directory where you saved the constraint file. Select the constraint file and click 'OK'. Click 'Finish' to add the XDC file to your project.

- At this point, both 'switch.v' and 'switch.xdc' should show up in the 'Sources' window. It is now time to create the hardware configuration for our specific FPGA and download the generated configuration to the ZYBO board.

- Select 'switch.v' in the 'Sources' window. In the 'Flow Navigator' under 'Program and Debug' panel, click on 'Generate Bitstream'. A warning will appear indicating 'No implementation results available'. Click 'Yes' to launch 'Synthesis and Implementation'. This will run all the processes necessary to create a bitstream, which can be downloaded to the FPGA. Running these processes may take several minutes; the spinning icon and output to the console indicate progress. You can check the progress in the 'Design Runs' panel located at the bottom of the screen. When a process completes, a appears next to the appropriate process name. Vivado follows these steps before creating the Bitstream File : synthesize the Verilog, map the result to the FPGA hardware, place the mapped hardware, and route the placed hardware.

- Once the bitstream generation is completed, we need to download the bitstream to the FPGA on the ZYBO board. Turn on the power to the ZYBO board and make sure that the jumper JP5 is set in 'JTAG' mode. In the 'Flow Navigator' window, under the 'Program and Debug' panel click on 'Open Hardware Manager'. Click on 'Open Target' and in the pop up select 'Open New Target' which will open the 'Open New Hardware Target' window. Click 'Next'. select 'Local Server' in the 'connect to' field. Click 'Next'. Select 'xilinx tcf' in Hardware Targets and 'xc7z010 1' in Hardware Devices as in (Figure 4). The 'arm dap 0' device is ARM Cortex Processor which is not needed for this lab. Click 'Next' and go through the summary and Click 'Finish'. Click on 'Program Device' under 'Hardware Manager' and select the FPGA 'xc7z010 1'. Click 'Program' to program the FPGA on the ZYBO board

- At this point, the FPGA should be programmed to function as described in 'switch.v'. Verify this by toggling the DIP switches 0 through 3 and observe LEDs 0 through 3. Demonstrate your progress to the TA.

- Implement a 4-bit counter using the LEDs (You do not need the switches for this exercise). The count value should update approximately every 1 second. Use the BTN0 and BTN1 push buttons on the ZYBO board to control the direction of the count. For example, when the BTN0 button is pressed, the counter should count up. Likewise when the button BTN1 is pressed, the counter should count down. When neither button is pressed, the count should remain the same. Demonstrate this operation to the TA upon completion.

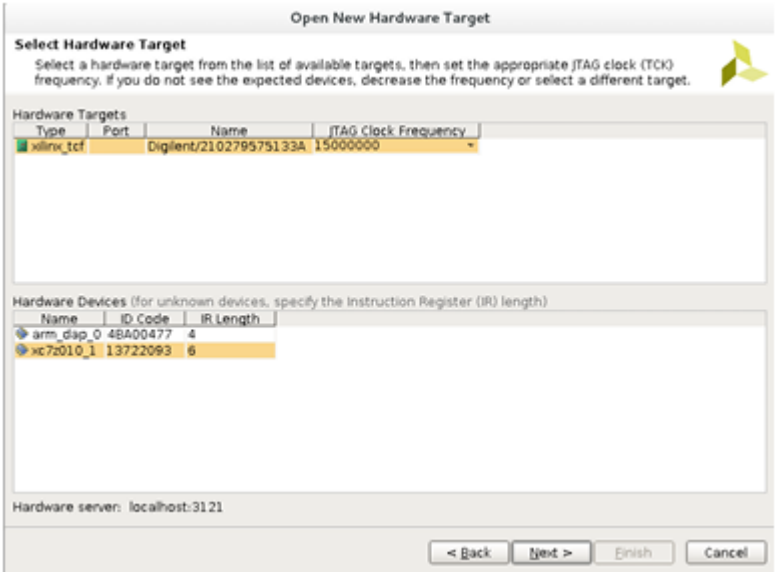


Figure 4: Hardware Target

Hints:

- Do not forget to add clock and reset as input pins to your verilog module.

- Skim through the user manual for the ZYBO board to determine the pin assignments form additional signals. The user manual may be found on the course website.

- After modifying the XDC to include the new ports (and removing unused ports) append the following text to the XDC:

set property PACKAGE PIN L16 [get ports CLOCK]
set property IOSTANDARD LVCMOS33 [get ports CLOCK]

- Note: The above XDC lines provide Vivado with timing constraints necessary to ensure proper design operation and assume your signal for clock is labeled 'CLOCK'.

- The L16 pin is the onboard clock with frequency 125MHz. Updates to the LEDs at this rate will not be visible, and thus, the incoming clock must be divided. Think back to your introductory digital logic class to determine how to divide a clock!

Evaluation of Results

Answer the questions and design the game described below.

- How are the user push-buttons wired on the ZYBO board (i.e. what pins on the FPGA do each of them correspond to and are the signals pulled up or down)? You will have to consult the Master XDC file for this information.
- What is the purpose of an edge detection circuit and how should it have been used in this lab?

Design a 'Jackpot game which works as follows: The LEDs glow in a one-hot fashion, which means that the LEDs are turned on one a time in a sequential manner. Get the transition to happen as fast as you can, while you can still make out which LED is on at a given of time. Assign a DIP switch to each of the LEDs. At any point in time, if you turn on the switch corresponding to the glowing LED, you win a Jackpot and all the LEDs start glowing!

Safety Precautions

References

- https://digilent.com/reference/programmable-logic/guides/getting-started-with-vivado
- https://digilent.com/reference/programmable-logic/zybo-z7/reference-manual
- https://digilent.com/reference/programmable-logic/guides/installing-vivado-and-vitis
- https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM
- https://digilent.com/reference/programmable-logic/guides/getting-started-with-vivado

9. Experiment 2 on ZYBOZ7 Development Board: Soft Microprocessor

Objective of the Experiment

The purpose of this lab is to familiarize you with Vivado by developing a software-based solution for controlling the LEDs like that which was done last week in pure FPGA hardware. To accomplish this goal, you will be guided through the process of creating a MicroBlaze processor system using the Vivado Block Design Builder. You will then add General Purpose Input/Output (GPIO) capabilities to the microprocessor via Intellectual Property (IP) hardware blocks from Xilinx. Finally, you will create software using the C programming language, which will run on the MicroBlaze processor in order to implement the appropriate LED functionality.

Theoretical Background

The Microblaze soft-core processor IP can be used to instantiate a processor within your FPGA design. This processor can be very useful for controlling and configuring hardware components. You can add a Microblaze processor and several useful components, including UART for standard output and DDR memory support, to your block design.

The microprocessor system you will build in this lab is depicted in the following figure. To the left of the diagram is the MicroBlaze soft IP processor. Connected to it are two Local Memory Buses (LMBs), iLMB and dLMB for instruction fetch and data access respectively. Each LMB has its own block RAM (BRAM) controller which provides the interconnect logic between the MicroBlaze and BRAM (local memory). The AXI interconnect connects the MicroBlaze (bus master) to peripherals (bus slaves) external to the microprocessor. Typically included in the list of peripherals are the debugger module. The debugger allows the SDK to interact with the MicroBlaze processor after the FPGA has been programmed. This is useful for initializing regions of memory outside of the FPGA and for general software debugging. The GPIO blocks provide the microprocessor with a means of controlling the LEDs and reading user input from the DIP switches and push buttons.

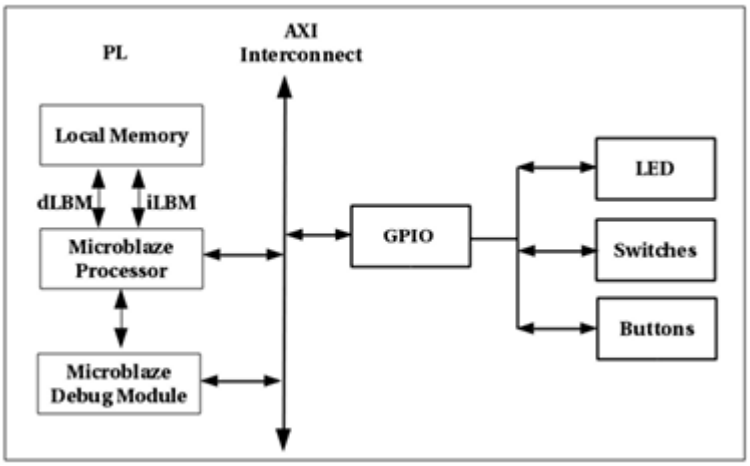


Figure 1: MicroBlaze System Diagram

Materials Used

- Zynq-7000 ARM/FPGA SoC Development Board
- USB Programming cables, USB UART cables, and Power Supply, as required by the board.
- Vivado and Vitis, the development environments

Procedure

- In the following steps, we will launch Vivado and create a block design. Before beginning, create a directory in your home folder for this lab. Try to avoid spaces and special characters in the

directory name as they have the potential for causing problems during the system build process. You may use the ‘mkdir’ command in an open terminal to create a directory.

- Next, type the following commands in the terminal window:
  >source /softwares/Linux/xilinx/Vivado/2015.2/settings64.sh
  >vivado
- The first command will set up the environment for Vivado, and the second command runs Vivado.
- Once Vivado launches, select ‘Create New Project’ and follow the same procedure as shown in Lab1 to create a new RTL project except do not add a source file in the project. In this lab you will use Xilinx Microblaze Processor.
- On the left side, in the Flow Navigator under the IP Integrator section, click on ’Create Block Design’. A window opens where you can specify the name of your design(eg. led sw). Leave the ’Directory’ and the ’Specify source set’ as they are. Click on the ‘OK’ button. (Figure 2)

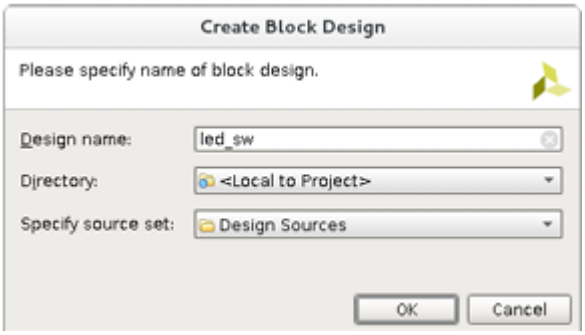


Figure 2: Create Block Design

- The Block Design Diagram opened (tab called Diagram) is empty. Right click within the dia gram tab and select ‘Add IP’. Search ‘MicroBlaze’ and double click on ‘MicroBlaze’ to add it to our design. Right click and select ’Run Block Automation’. Select the following configuration for the MicroBlaze processor and click ‘OK’. (Figure 3)

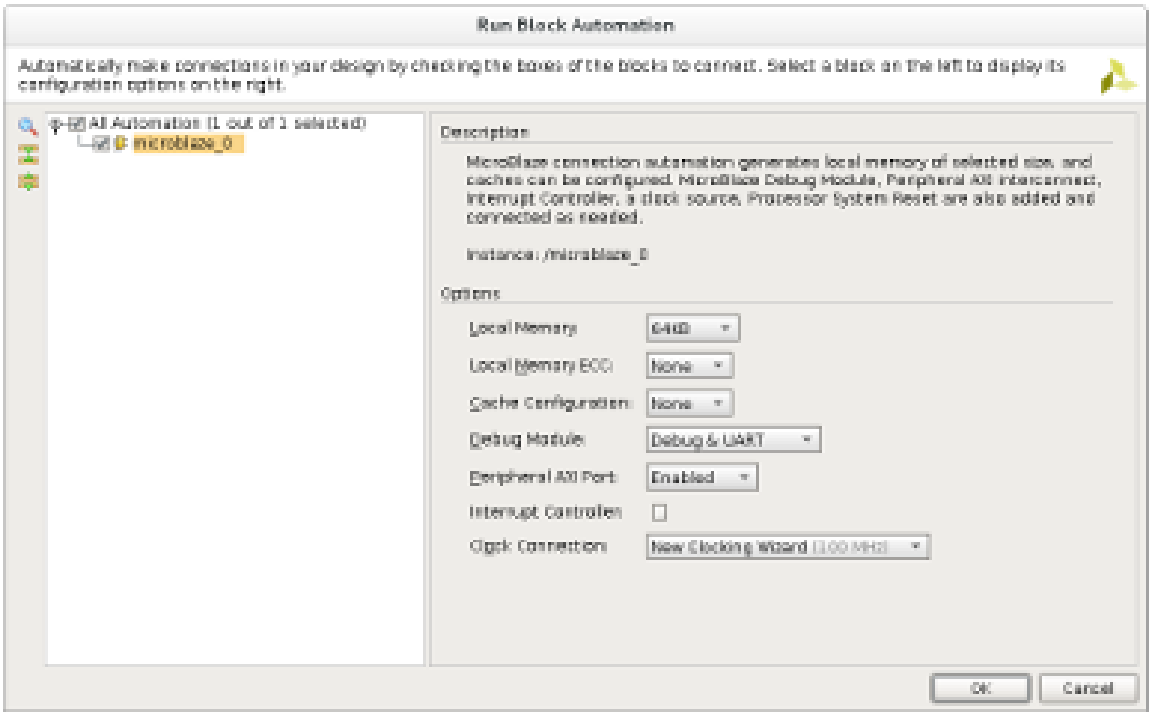


Figure 3: Run Block Automation

Local Memory: 64KB
Local Memory ECC: None

Cache Configuration: None Debug Module: Debug & UART
Peripheral AXI Port: Enabled
Interrupt Controller: Unchecked.
Clock Connection: New Clocking Wizard (100 MHz)

- Once the Diagram is generated, double-click on the 'Clocking Wizard' block named 'clk wiz 1'. This launches the 'Re-customize IP' window. Select 'Clocking Options' and change the source of the 'Primary Input Clock' from 'Differential clock capable pin' to ' Single-ended clock capable pin'. Click 'OK'.

- Right-click and select 'Run Connection Automation' in the diagram. In the 'Run Connection Automation window', check 'All Automation' and click 'OK'.

- The processor is set up, now we need to add General Purpose IO (GPIO) blocks to interact with LEDS, Switches and Buttons on the ZYBO board. Right click and select 'ADD IP'. Search for 'GPIO' and select 'AXI GPIO' to add it to the design. Double click on the GPIO block named 'axi gpio 0' and select the 'IP configuration' tab. Set the following configuration for the GPIO and click 'OK' (Figure 4).

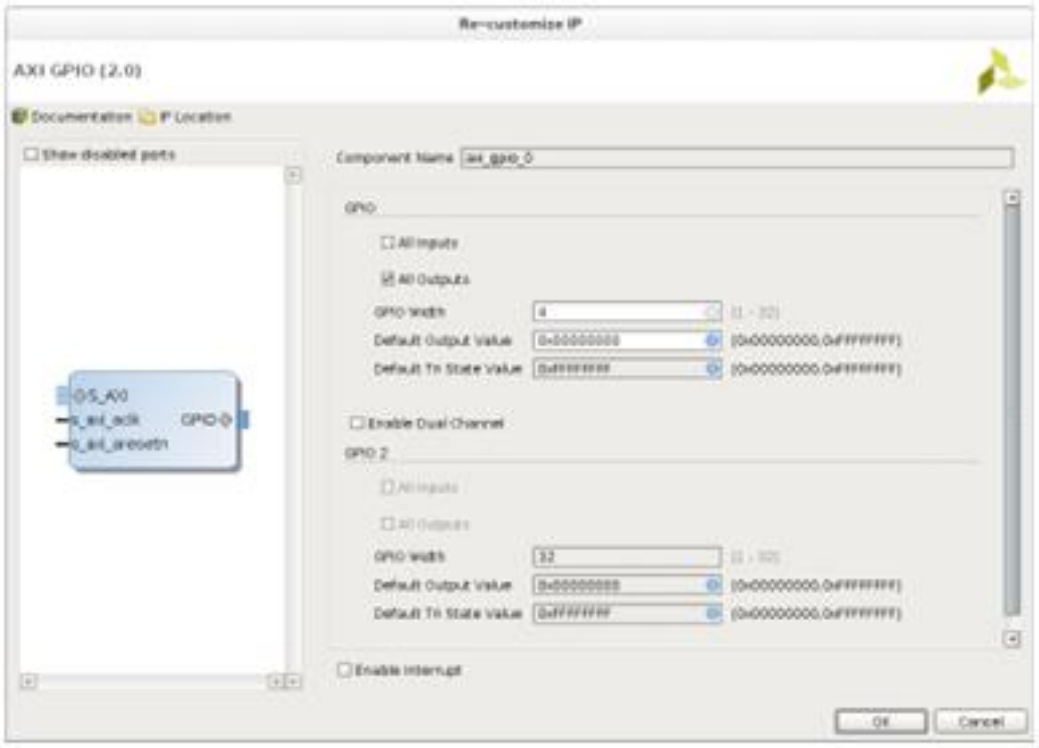


Figure 4: Customize GPIO

All Inputs: (not checked)
All Outputs (checked)
GPIO Width: 4
Leave the remaining values unchanged.

- Right click and select 'Run Connection Automation', check 'All Automation', and click 'OK'. Now design is connected. To make it look neat, right click and select 'Regenerate Layout

- You are using the GPIO block to configure LEDs, so let's rename the 'AXI GPIO' block. Select the 'AXI GPIO' block and rename the block to 'led' in the 'Block Properties' panel and press 'Enter'.

Repeat the same procedure for the port connected to the ‘AXI GPIO’ block and rename it to ‘led’. The layout should look like Figure 5

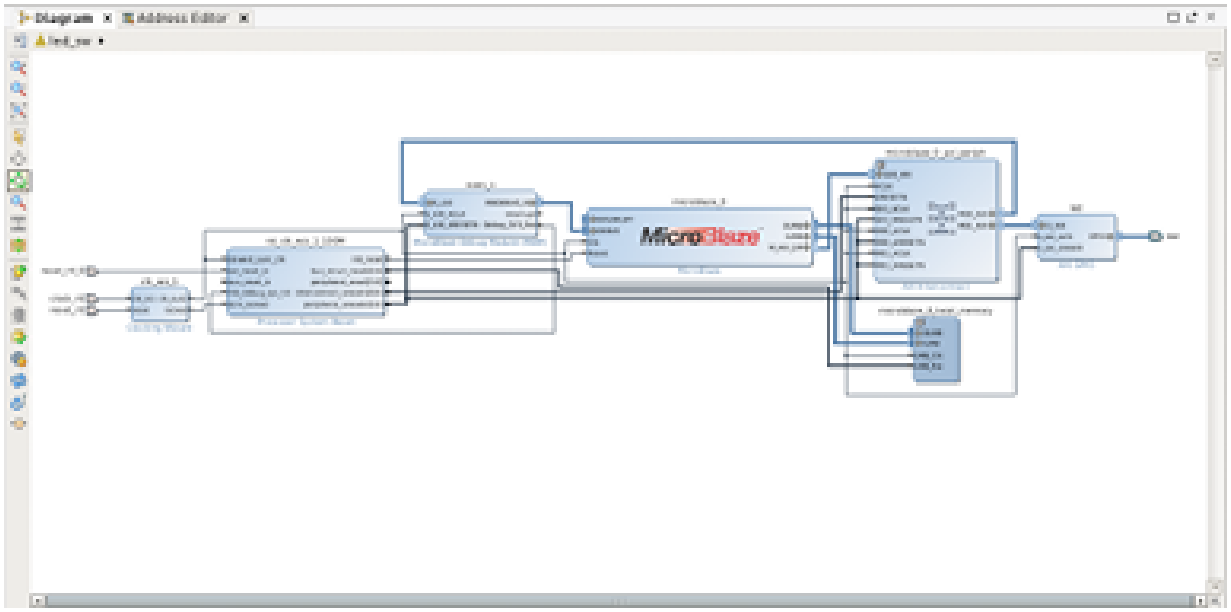


Figure 5: Layout

- Double click the ‘reset rtl 0’ port in the diagram. You can see that it is ‘ACTIVE LOW’. Double Click on the ‘reset rtl’ port and observe that it is ‘ACTIVE HIGH’.

- These are the reset ports for Microblaze. On ZYBO board we have dedicated reset pins for reset. Refer to manual and see the functionality of BTN6 and BTN7.

- These two reset ports can be configured to a push button on the board. For the purpose of this lab, connect the ports to constant values and use the dedicated pins on board for reset.

- Select ‘Constant’ IP from the IP repository (Right click and ADD IP) and add two instances to the diagram. Double click on one of the constant IP blocks and set ‘Constant Width’ to 1 and ‘Constant Value’ to 1. Rename this block to ‘VDD’. Now, right click on the ‘reset rtl 0’ port and select ‘Delete’ to delete it. Connect the constant block to ‘ext rst in’ of the ‘Processor System Reset’ because it is ‘ACTIVE LOW’. Connect the constant block by clicking on the output of the constant block and dragging it to the ‘ext_rst in’ pin.

- Repeat the same procedure for the other constant block but set ‘constant value’ to 0 and rename it as ‘GND’. Delete the ‘reset rtl’ port and connect it to the ‘reset’ port of the ‘Clocking Wizard’.

- Right click on the diagram and select ‘Regenerate Layout’. The final layout should look like Figure 6.

In the following steps we map the IO ports to the LEDs and buttons.

- In the design tab, expand ‘External Interfaces’ and ‘Ports’ and examine the ports listed. When ‘led’ is expanded it shows the ‘led tri o’and ‘clock rtl’ ports listed. We need to connect these ports to the ZYBO board.

- In the sources panel, right click on the constraints and select ‘Add sources’. Next, in the add sources window, select ‘Add or create constraints’ and click ‘Next’. Click on the green ‘+’ button

and select ‘create file’. The ‘Create Constraints File’ window will open, give a file name (eg. led), and click ‘OK’. Select ‘Finish’ to create a constraint file.

- In the sources panel, expand ‘Constraints’ and double click on the created XDC file to open it.

- Add the following code to attach the above ports to the pins on ZYBO board.

```
#clock_rtl
set_property PACKAGE_PIN L16 [get_ports clock_rtl]
set_property IOSTANDARD LVCMOS33 [get_ports clock_rtl]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clock_rtl]

#led_tri_o
set_property PACKAGE_PIN M14 [get_ports {led_tri_o[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_tri_o[0]}]

set_property PACKAGE_PIN M15 [get_ports {led_tri_o[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_tri_o[1]}]

set_property PACKAGE_PIN G14 [get_ports {led_tri_o[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_tri_o[2]}]

set_property PACKAGE_PIN D18 [get_ports {led_tri_o[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_tri_o[3]}]
```

In the following steps we will Generate Bitstream and launch the SDK.

- Right click in the diagram and select ’Validate Design’ to check for any errors in the design. If everything is correct, you have successfully built the hardware platform.

- Right click on your project in the sources panel and select ‘Create HDL wrapper’. Select ‘Let Vivado manage wrapper and auto update’ and click ‘OK’ to create the top level module for the design. Finally, click ‘Generate Bitstream’ to create the program file. Once this is done, we will be able to export our hardware design and switch over to the SDK to program the FPGA and run the program to manage LEDs.

- If any errors are found please rectify them. Once successfully completed, select File ‘Export Hardware’. Check ‘Include bitstream’ to export the hardware platform.

- Now, launch the Software Development Kit(SDK) using ‘File’ ‘Export’ ‘Launch SDK’. Select ‘local to project’ as the location.

In the following steps we will use the SDK to create the software application.

- Once the SDK is open. Select ‘File’ ‘New’ ‘Application Project’. Give a project name (eg.counter sw). Click ‘Next’ and select the ‘Empty Application’ and click ‘Finish’.

- You will now have three files in the ’Project Explorer’ panel (counter sw, counter sw bsp and led wrapper hw platform 0).

- Using your favorite editor, create a file called ‘lab2a.c’. Type the C code shown below in your f ile and save it in your lab2 directory

```
#include <xparameters.h>
#include <xgpio.h>
```

```
#include <xstatus.h>
#include <xil_printf.h>

/* Definitions */
#define GPIO_DEVICE_ID XPAR_LED_DEVICE_ID   /* GPIO device that LEDs
are connected to */
#define WAIT_VAL 10000000

int delay(void);

int main()
{
    int count;
    int count_masked;
    XGpio leds;
    int status;

    status = XGpio_Initialize(&leds, GPIO_DEVICE_ID);
    XGpio_SetDataDirection(&leds, 1, 0x00);

    if (status != XST_SUCCESS) {
        xil_printf("Initialization failed\n");
    }

    count = 0;

    while (1)
    {
        count_masked = count & 0xF;
        XGpio_DiscreteWrite(&leds, 1, count_masked);
        xil_printf("Value of LEDs = 0x%x\n\r", count_masked);
        delay();
        count++;
    }

    return 0;
}

int delay(void)
{
    volatile int delay_count = 0;
    while (delay_count < WAIT_VAL)
        delay_count++;
    return 0;
}
```

- Look through the code you just wrote and try to understand what exactly is going on. Notice we include the ‘xparameters.h’, ‘xstatus.h’, and ‘xgpio.h’ header files. Open up these files from the ‘Outline’ panel to the right of the window and understand what they provide. At the end of the lab, you will find questions on these files.

- In the project explorer, expand ‘counter sw’, right click on ‘src’ and select ‘import’. In the import window expand ‘General’, select ‘File System’, and click ‘Next’. Click ‘Browse’ and select the folder where you saved the lab2a.c file and click ‘OK’. Select the lab2a.c file from the import

window and click ‘Finish’.

Simulation

- Now we have the software application ready. Connect the FPGA and switch it ON. In the SDK, click ‘Xilinx Tools’ and select ‘Program FPGA’. Next, the ‘Program FPGA’ window will appear, and select ’Program’ to program the bitstream file generated in Vivado onto the FPGA. A warning will appear indicating that there is no PS in the design. PS is short for processing system, which represents the ARM Cortex Processor in the ZYBO board. In this lab, the Microblaze processor is used, which is implemented completely on the FPGA, and hence it is called a soft processor. Click ‘OK’ on the message to program the FPGA.
- The next step is to run the C application on the system. Click ‘Run’, and select ‘Run Configuration’. In the ‘Run Configuration’ window , select ‘Xilinx C/C++ Application(GDB)’. Click on the icon that resembles a document with a ‘+’ sign on top left corner of the window to create a configuration file. Name the configuration (eg. led sw) and in the application tab, select ‘Browse’ in the Project Name field and select the ‘counter sw’. In the STDIO connection, check ‘Connect STDIO to Console’. Select ‘JTAG UART’ as Port. Click on ‘Apply’ and select ‘Run’ to deploy the program to the FPGA.

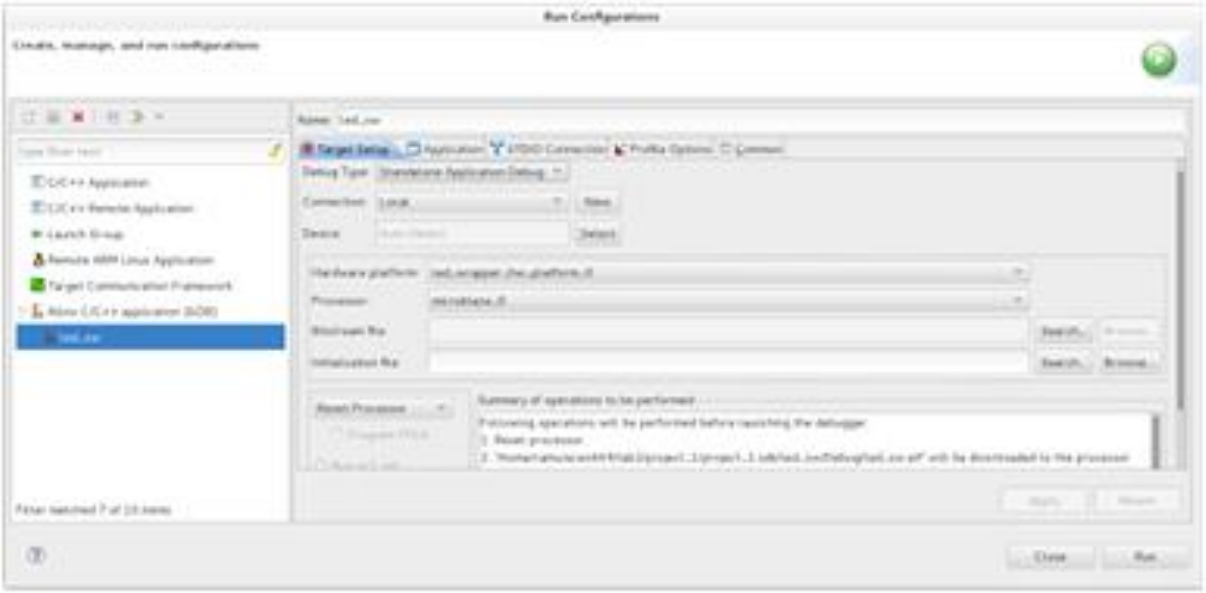


Figure 7: Run Configuration

- If everything is correct, you should now be able to see the LEDs glowing in the order from 0 to 15 and the output from the printf statements on the console in the SDK. NOTE: For the purpose of debugging stop the program by entering ‘stop’ on the XMD console and re-run the program. The XMD console can be found under ‘Xilinx Tools’.

Evaluation of Results

- All you have done is to see the LEDs glowing. If you succeed, then explain how the C programs run on FPGAs.
- What is ‘Soft Microprocessors’?
- How many CBS blocks are used for MicroBlaze? or Is any CBS block needed?

Safety Precautions

References

- https://digilent.com/reference/programmable-logic/guides/getting-started-with-vivado
- https://digilent.com/reference/programmable-logic/guides/getting-started-with-ipi

10. Experiment 3 on ZYBOZ7: Creating a Custom Hardware IP and Interfacing it with Software

Objective of the Experiment

The purpose of the lab this week is to familiarize you with the process of creating and importing a custom IP module for a Zynq Processing System based system. We will be using the 'Create and Package IP' in Vivado to develop a custom peripheral for performing integer multiplication. We will then integrate the integer multiplication peripheral into a microprocessor system and develop software to interact with the peripheral using the SDK. This lab serves as a simple hardware/software co-design example.

Theoretical Background

The microprocessor system you will build in this lab is depicted in Figure 1. In the previous lab a soft processor Microblaze was used, however in this lab you will use the ARM Cortex A9 processor in the Zynq Chip on ZYBO board. The Zynq chip is divided into Processing System(PS) and Programming Logic(PL). PS has a dual-core ARM Cortex-A9 processor and PL uses Xilinx 7 series FPGA logic cells. In place of the GPIO modules, utilized in the last lab, is a multiplication block, which represents the integer multiplication peripheral you will create. The UART in Figure 1 will be used to connect the USB-UART port(J11) on the ZYBO board to the workstation computer via a cable which will display the output of the software executing on the PS. The software you will develop in this lab will provide proof of operation for your multiplication peripheral.

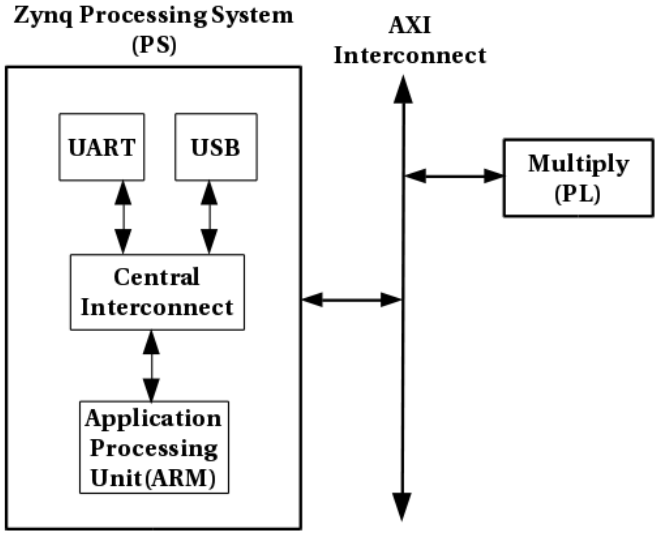


Figure 1: Zynq System Design

Materials Used

- Zynq-7000 ARM/FPGA SoC Development Board
- USB Programming cables, USB UART cables, and Power Supply, as required by the board.
- Vivado and Vitis, the development environments

Procedure

Create Zynq Base System

- To begin, open vivado and create a new project as shown in previous lab with one exception. In the Default part window, click on 'Boards' and select 'Zybo' as shown in Figure 2. As discussed in Lab 1 we will select hardware using the 'Board' tab from now on.

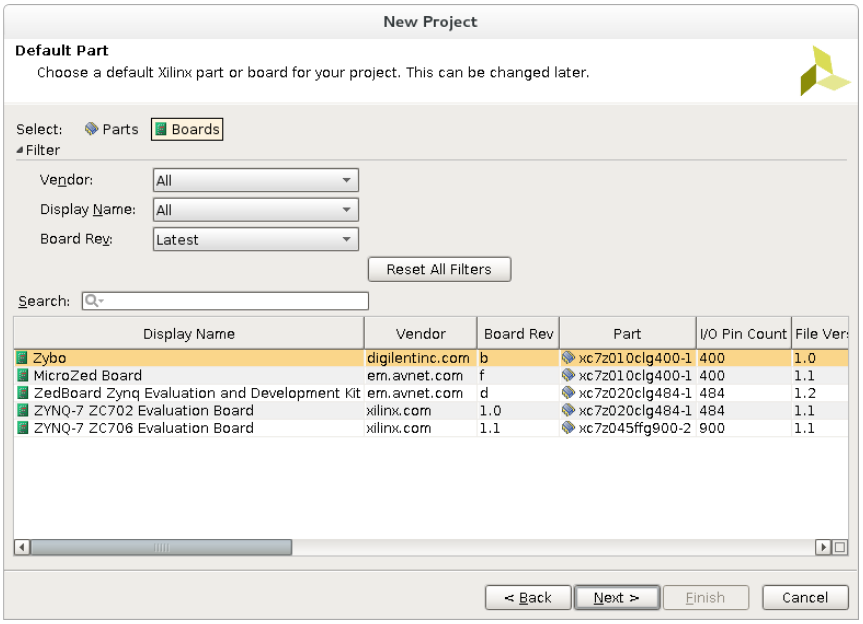


Figure 2: ZYBO Board

- Click on 'Create Block Design' and name the design 'multiply'. In the diagram, add 'ZYNQ7 Processing System' IP as Figure 3.

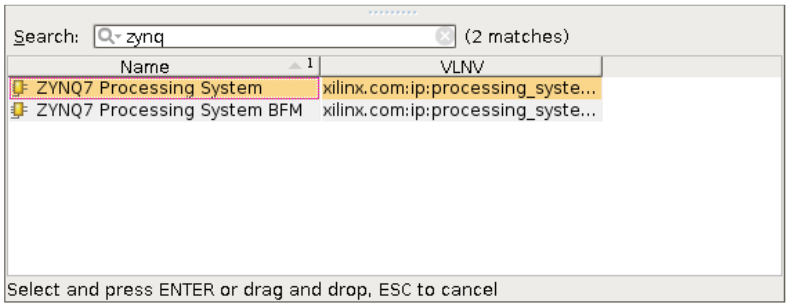


Figure 3: Zynq Processing System

- Do not select 'Run Block Automation' as in previous lab. Double click on the PS IP to open 'Re-customize IP' window. Download the 'ZYBO zynq def.xml' file from home/faculty/shared/ECEN449 and save this XML file somewhere under your Linux account by using the 'cp' command. In the 'Re-customize IP' window, click on 'Import XPS Settings', and import the XML file you just downloaded. Then click on 'Peripheral I/O Pins' tap, and uncheck all the peripheral I/O pins.

- In the 'Re-customize IP' window, click on 'Peripheral I/O Pins'. To build the hardware depicted in Figure3. Enable 'UART 1' by clicking on the check box. Double check and make sure that only the 'UART 1' is selected.

- Click on 'Zynq Block Design' tab and observe the diagram. Note that the 'Application processing Unit' which represent the ARM processors on the ZYBO board is connected to 'UART 1' through a 'Central Interconnect' block. Click 'OK'

- PS is ready and the next part is to create the 'multiply' IP. Click on 'Tools' and select 'Create and Package IP'. This will open 'Create and Package New IP' window and click on 'Next'. Now select 'Create a new AXI4 peripheral' and click on 'Next'.

- A window will appear prompting you to assign a name and version number to your peripheral (Figure 4). Name the peripheral ‘multiply’ and leave the version number as default (i.e. 1.0). You can enter a short description of your peripheral if you would in the ‘Description’ field.

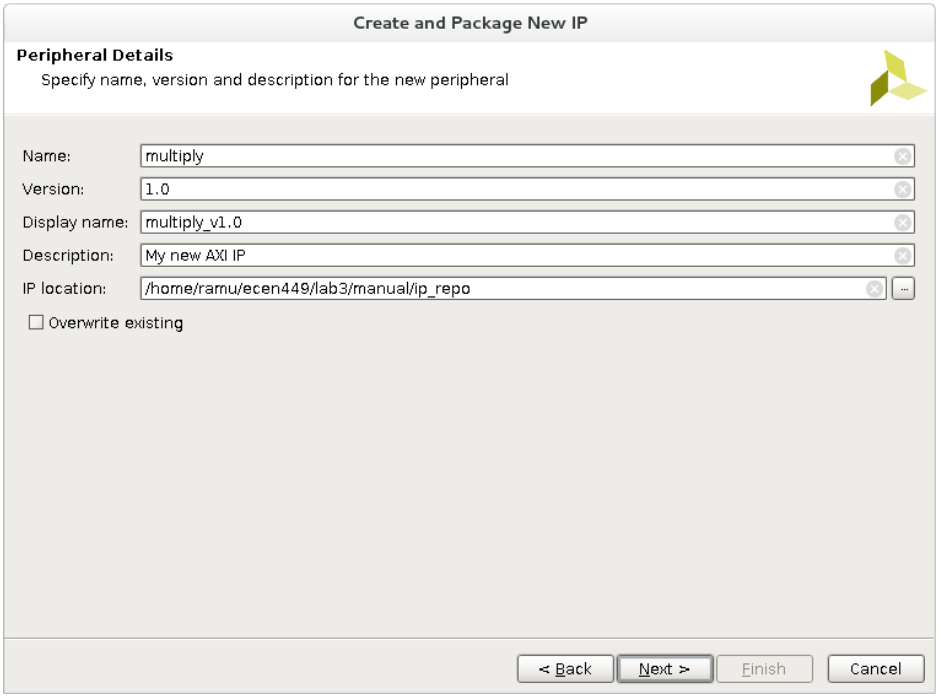


Figure 4: Create New IP

- The next window will ask us to select ‘Interface’. Leave the default values
  Interface type: Lite
  Interface mode: slave
  Data Width: 32
  Number of Registers: 4
  We will need only three registers for the ‘multiply’ IP however the minimum number of registers allowed is 4 (see Figure 5, Click on ‘Next’. You will see a summary of the IP we have created. We still need to add the functionality of the multiplier to this IP. Check ‘Edit IP’ and click ‘Finish’. Another Vivado window will open which will allow you to modify the peripheral that we created.

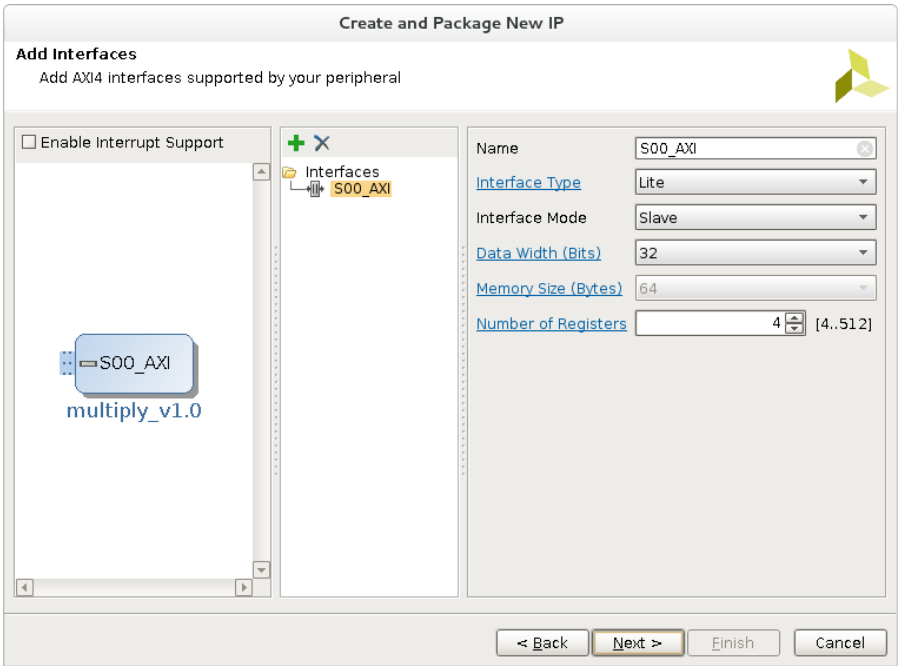


Figure 5: IP Interface Operations

- At this point, the peripheral that has been generated by Vivado is an AXI lite slave that contains 4 x 32 bit read/write registers. We want to add our multiplier code to the IP and modify it so that two of the registers connect to the multiplier inputs and another register connects to the multiplier output.

Edit 'multiply' IP and Import it to PS

- Open the new Vivado window that contains the peripheral we created (not the base project). Review the information under the various tabs in the Package IP tab.

- In the sources window, expand 'multiply_v1_0' and double click on the 'multiply_v1_0_S00_AXI.v' file to open it.

- Examine the Verilog code in file and try to understand each function, especially how the 'read' and 'write' functions are implemented. Comment out and code that writes to 'slv_reg2' (i.e. 'slv_reg2<='). This will deactivate the write capabilities to the third software register which is our multiplier output. Remember that we cannot have two drivers for a particular register unless we add multiplexing logic.

- While the 'multiply_v1_0_S00_AXI.v' file is still open, locate the

```
'// Add user logic here' line and insert the following code:
      reg [0 : C_S_AXI_DATA_WIDTH-1] tmp_reg ;
      always @( posedge S_AXI_ACLK ) begin
            if ( S_AXI_ARESETN == 1'b0 ) begin
                  slv_reg2 <=0;
                  tmp_reg <=0;
            end
            else begin
                  temp_reg <= slv_reg0 * slv_reg1 ;
                  slv_reg2 <= temp_reg ;
            end
      end
```

- In Package IP tab, click on ‘Review and Package’ and select ‘Re-package’. Now Vivado will repackage the IP with added functionality. When Vivado finishes packaging the IP, close the project.

- At this point in the lab, we have used Vivado to generate template hardware peripheral files for our multiplication peripheral based on our specifications. We have also added user logic in Verilog to our template for the multiplication functionality of our peripheral. We are now ready to import our peripheral into Vivado and add it to our PS system.

- Now, select the Vivado window which has the PS system. Open the diagram tab and add ‘multiply’ IP to the PS system. Select ‘Run Connection Automation’ and in the prompted window select ‘All Automation’ and click ‘OK’. Once automation is completed, a layout is generated. Right click on the diagram and select ‘Regenerate Layout’ to re draw the layout design. The layout is shown in Figure 6. A ‘Processor System Reset’ block is also created which synchronizes the peripheral and interconnect reset to clock.

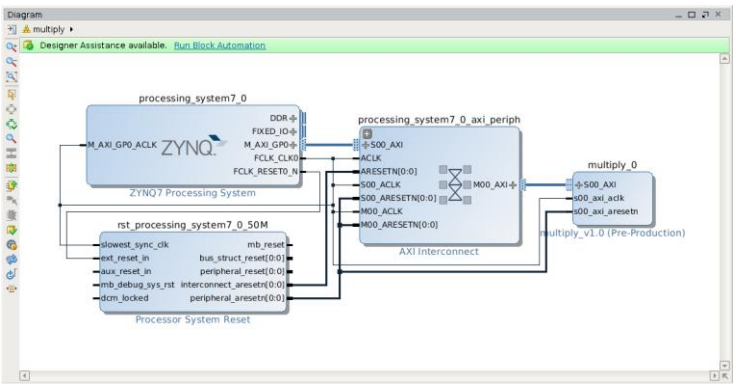


Figure 6: Layout of the Design

- In the source tab, right click on your block design and select ‘Create HDL wrapper’. In the ‘Create HDL wrapper’ window select ‘Let Vivado manage wrapper and auto update’. This will create the top module for the blocks in the block diagram. Click ‘OK’.

- Now we have the PS system and multiply IP ready. It is time to generate the bitstream. In the Flow Navigator, select ‘Generate Bit Stream’. Ignore any critical messages during the process.

- Once the bit generation is complete, it is time to write an application and test the multiply IP.

- Export the design, including bit stream, as shown in the previous lab.

Simulation

Launch SDK and write a multiplication test application.

- Currently, our hardware should be ready to go. Next step is to create an application to test our ‘multiply’ IP peripheral. Open the SDK and click on ‘File’ and select ‘New’ -> ‘Application Project’. In the new project window, give a name (eg. multiply test) to the project and leave the default values for the remaining fields. Click ‘Next’ and select ‘Hello World!’ and ‘Finish’. This step will generate the necessary templates files for our PS on the ZYBO board.

- In the project explorer, expand ’multiply test’. Under the src folder, open ‘helloworld.c’ file and examine the code generated.

- Edit the 'helloworld.c' source file to write values to the registers 'slv reg0' and 'slv reg1' and read the multiplication result from 'slv reg2'. The value stored in each of these three registers should be printed to the terminal using printf.

- Once you have written the source file, save it under src folder. Click on 'Xilinx Tools' and select 'Program FPGA' to program the bitstream file to the ZYBO board. Click 'Program'. Under 'multiply test' expand 'binaries'. Right click on 'multiply test.elf', select 'Run As' and click on 'Launch on Hardware(GDB)'. This command creates a configuration file which we can use to run our application. Vivado will launch the application on the ZYBO board.

- To see the output of the printf statements in our code, we must use 'picocom', a serial console application on the CentOS machines. Open a terminal window and type the following:

  ```
  $ source /softwares/Linux/xilinx/Vivado/2015.2/settings64.sh
  $ picocom -b 115200 /dev/ttyUSB1
  ```

  If everything is correct, you will see text being printed to the serial console. Demonstrate your progress to the TA.

The following Hints will help:

- You will need to include the xparameters.h and multiply.h in the source file. SDK will dislay the files included in your source code in the outline window to the right side.
- Look for functions to read and write to a register in multiply.h.
- The RegOffset value for 'slv_reg0' is 0, and the four 32-bit registers are consecutively located in memory.
- Use a for-loop to write different values (varying from 0 to 16) in 'slv_reg0' and 'slv_reg1' and read the corresponding multiplication result from 'slv_reg2'.
- To read or write to slave registers, make sure that the base address is of the type 'Unsigned 32-bit integer (u32)'
- Do not remove the functions that initialize and clean the board in your C program. They are essential for your code to operate properly.
- If you close the multiply IP block, you can open it again by right clicking on the multiply IP block and selecting 'Edit in IP Packager'.

Evaluation of Results

In this experiment, we successfully designed a custom hardware IP core (a simple integer multiplier) and integrated it into a Zynq-based system using Vivado. The process included creating an AXI4-Lite slave interface, implementing the multiplication functionality in Verilog, and wrapping the logic into a reusable IP block. This custom peripheral was then instantiated in a block design that included the ARM Cortex-A9 processor and UART interface, and communication between hardware and software was established using Xilinx SDK.

Through this lab, we gained hands-on experience in hardware/software co-design, understanding how to transition from RTL-level hardware design to system-level integration. The project highlighted the power of AXI interfacing and modular IP development, which is a cornerstone of FPGA-based embedded system design.

In real-world applications, custom IPs such as this multiplier could be tailored for signal processing, cryptographic operations, or AI accelerators any scenario where offloading computation from software to hardware yields speed and power efficiency benefits.

- What is the purpose of the tmp_reg from the Verilog code provided in lab, and what happens if
- What values of ‘slv_reg0’ and ‘slv_reg1’ would produce incorrect results from the multiplication block? What is the name commonly assigned to this type of computation error, and how would you correct this? Provide a Verilog example and explain what you would change during the creation of the corrected peripheral.

Safety Precautions

References

- https://digilent.com/reference/programmable-logic/guides/getting-started-with-vivado
- https://digilent.com/reference/programmable-logic/zybo-z7/reference-manual
- https://digilent.com/reference/programmable-logic/guides/installing-vivado-and-vitis
- https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM
- https://digilent.com/reference/programmable-logic/guides/getting-started-with-vivado

11. Experiment 4: Linux boot-up on ZYBO board via SD Card

Objective of the Experiment

The purpose of lab this week is to get Linux up and running on the ZYBO board. There are many advantages to running an Operating System (OS) in an embedded processor environment, and Linux provides a nice open-source OS platform for us to build upon. This week, you will use Vivado to build a Zynq (ARM Cortex A9) based microprocessor system suitable for running Linux, and you will also use open source tools to compile the Linux kernel based on the specification of your custom microprocessor system. You will then combine the bit stream with FSBL(First Stage Boot Loader) and u-boot(Universal Boot Loader) to create Zynq Boot Image. FSBL initialize the Processing System(PS) with configuration data and initializes u-boot. u-boot is the boot loader that holds the instructions to boot the Linux Kernel. we need a RAMDISK, a temporary file system that is mounted during Kernel boot. The boot loader needs a device tree which has information about the physical devices in the system and this information is stored in 'device tree blob'. We will use all these files to boot Linux on ZYBO board using a SD card.

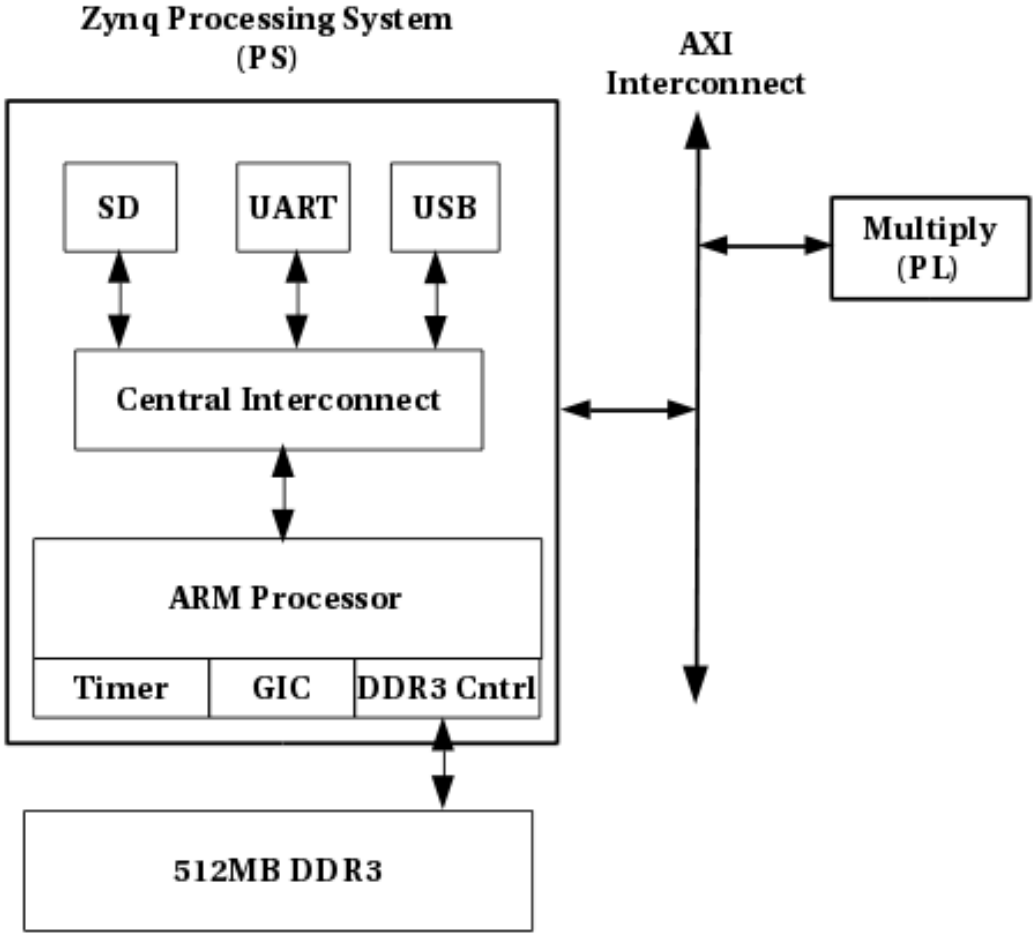


Figure 1: Zynq System Diagram

Theoretical Background

The microprocessor system you will build in this lab is depicted in Figure 1. As in last lab, this system has a Zynq processor, an UART Peripheral, and a custom multiplication peripheral. Unlike last lab, however, this system has an SD Card, a timer and a DDR3 (Double Data Rate v3 Synchronous Dynamic Random Access Memory) controller. Additionally, the PS itself has a Memory Management Unit (MMU) and instruction and data cache. These additional peripherals along with the MMU within the PS are required to run Linux. The SD card controller provides sd card read/write access. The DDR3 SDRAM controller

provides the system with 512MB of RAM where the Linux kernel can reside. The interrupt controller enables interrupt handling necessary for interaction with I/O. The MMU within the PS enables virtual memory, which is required to run a mainstream Linux kernel. The PS(ARM Cortex A9) cache is added to improve performance, as DDR SDRAM accesses have high latency. Interrupts to the processor are addressed by Generic Interrupt Controller(GIC). The timer is necessary for certain OS system calls.

Materials Used

- Zynq-7000 ARM/FPGA SoC Development Board
- USB Programming cables, USB UART cables, and Power Supply, as required by the board.
- Vivado and Vitis, the development environments
- SD Card with Linux Image

Procedure

To begin, we must create a base PS system which includes all the peripherals shown in Figure 1.

- Create a new directory by name Lab4, Open Vivado and create new project as explained in Lab 3. Make sure to select ZYBO board in part window.

- Next create block design, add 'ZYNQ7 Processing System' (PS) IP, import 'ZYBO_zynq_def.xml' file' as shown in the previous lab.

- (c) Run block automation.

- Now double click on PS IP, it will open 'Re-customize IP' window. Go to 'Peripheral I/O pins' tab and observe the peripherals that are enabled in the system.

- Enable SD 0, UART 1 and TTC 0 peripherals by clicking the tick mark on the corresponding peripheral and disable the remaining peripherals.

- Copy 'ip_repo' folder which contains 'multiply' IP from Lab 3 and paste into Lab 4 directory.

- Click on 'Project Settings' under project Manager tab. Click on 'IP' in the project setting window. Click on the green '+' sign and add the 'ip_repo' directory as shown in Figure 2. Click apply to add multiply IP to the current project IP catalog. Now you can add multiply IP to the base system.

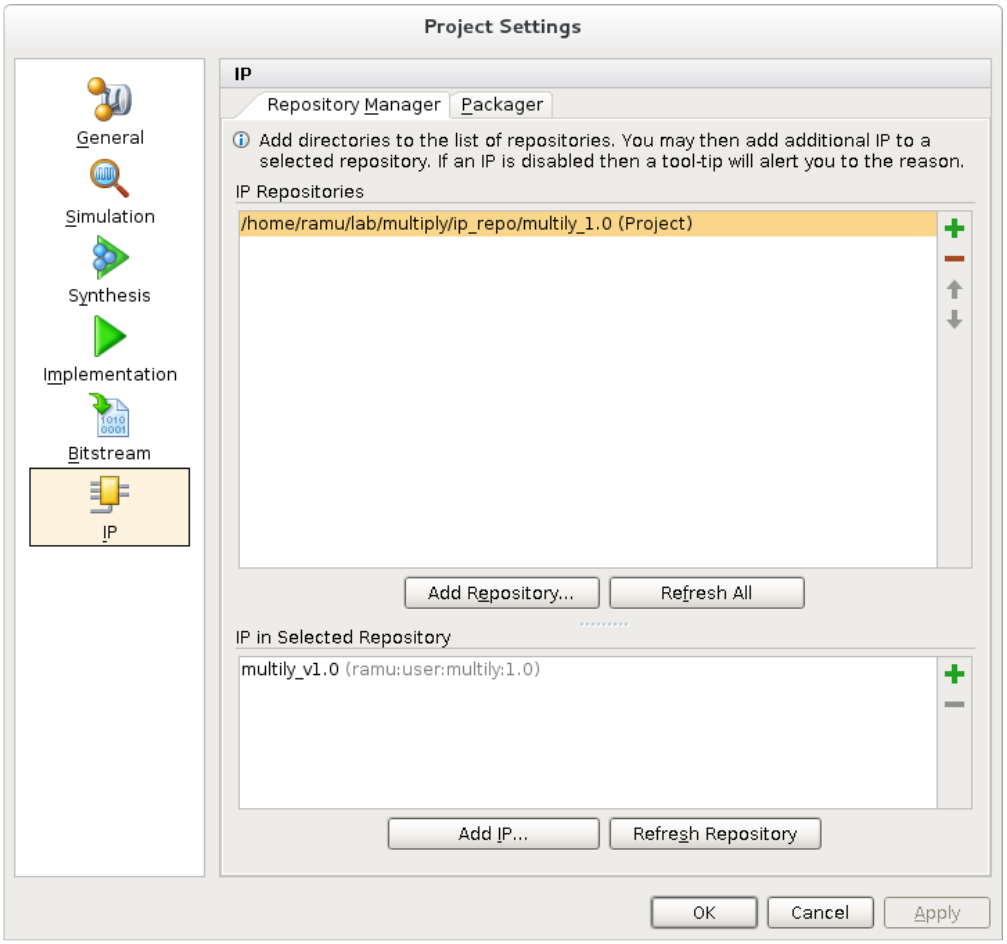


Figure 2: Add IP repository

- Add multiply IP to the base system.

- Run connection automation as shown in the previous lab. Now the base system should look like the one as in Figure 3.

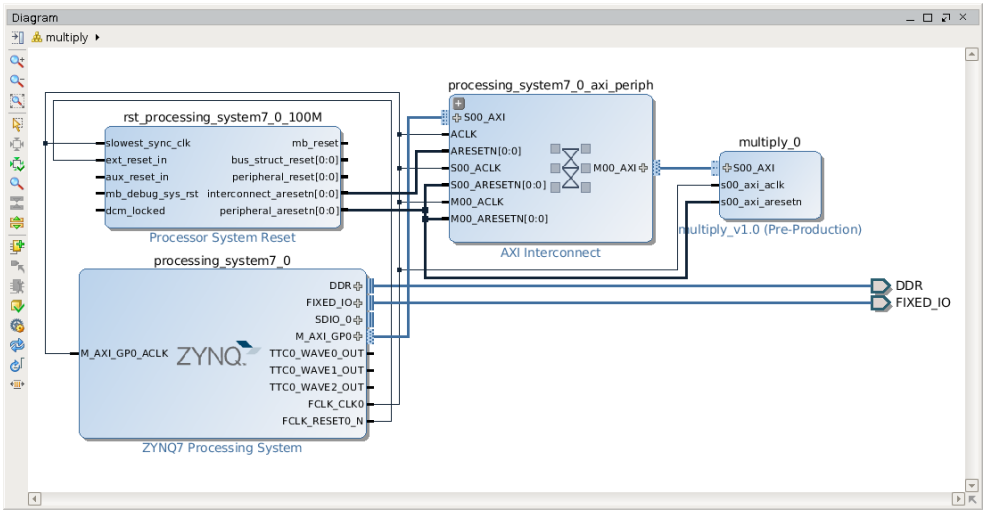


Figure 3: Base System Desing

- Create HDL wrapper to the base system as shown in the previous lab.
-
- Click on 'Generate Bitstream' to create the bit stream file. Now that we have bit stream, next is

to compile the u-Boot file.

u-boot

- Untar the '/homes/faculty/shared/ECEN449/u-boot.tar.gz' file by executing the following command from within your lab directory
  >tar -xvzf /homes/faculty/shared/ECEN449/u-boot.tar.gz

- To compile U-Boot, we need cross-compile tools which are provided by Vivado . Those tools have a prefix arm-xilinx-linux-gnueabi- to the standard names for the GCC tool chain. The prefix references the platforms that are used. In order to use the cross-platform compilers, make sure the Vivado 2015.2 settings have been sourced.

- Since this is an universal boot loader we need to configure it to the target device, in our case our ZYBO board. To configure and build U-Boot for ZYBO, run the following commands in u-boot directory:
  >make CROSS_COMPILE=arm-xilinx-linux-gnueabi- zynq_zybo_config
  NOTE:make sure to source settings64.sh before executing command

- You should see the terminal message 'Configuring for zynq_zybo board...'

- Now we have configured u-Boot, to compile run the following command in terminal.
  make CROSS_COMPILE=arm-xilinx-linux-gnueabi-

- After the compilation, the ELF (Executable and Linkable File) generated is named u-boot. Add a .elf extension to the file name so that Xilinx SDK can read the file layout and generate boot.bin. The u-boot file can be found under u-boot folder.

- Next step is to generate boot.bin. Open Vivado and export bitstream to SDK as shown earlier in Lab 3. Make sure to select 'include bitstream' while exporting the design.

- After SDK launches, the hardware platform project is already present in Project Explorer on the left of the SDK main window. We now need to create a First Stage Bootloader (FSBL). Click File->New->Application Project.

- In the new project window, name the project as FSBL and click on Next, in the Templates window select 'Zynq FSBL' and click on Finish.

- Click on Project and select 'Build All' to build the project.

- Now we have all of the files ready to create BOOT.BIN. Click Xilinx Tools -> Create Zynq Boot Image.

- In the Create Zynq Boot Image window (as shown in Fig. 4), Click Browse to set the path for output.bif. Give the path to your lab4 directory. Figure 4: Zynq Boot Image FSBL as the name suggests it should be the first file in the boot image. Click add in the boot partitions section to add FSBL.elf. In the 'add partition' window, select 'Browse' and FSBL can be found under the directory lab4/project_name.sdk/FSBL/Debug/. Make sure the partition type is set as bootloader. This will indicate FSBL as the initial boot loader. Select 'OK'

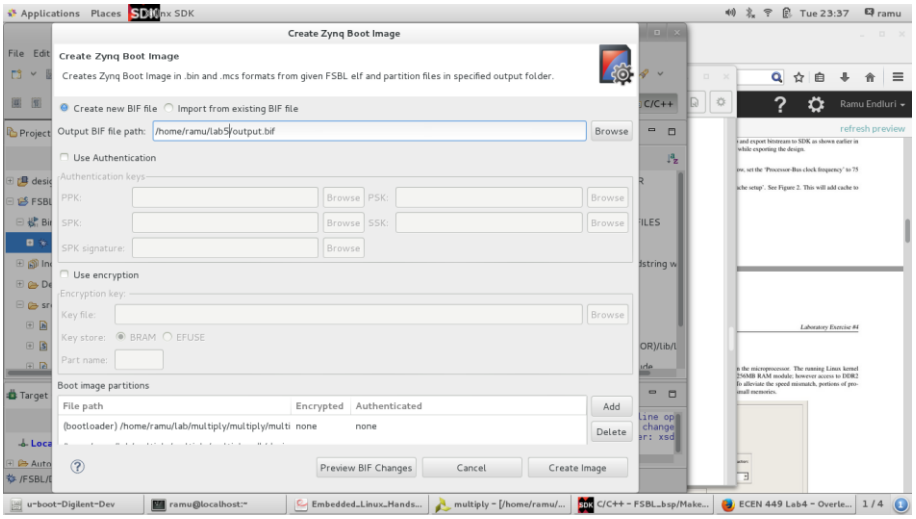


Figure 4: Zynq Boot Image

- Next step is to add the bitstream generated. Bitstream(.bit file) can be found under project folder at /project_name.sdk/design_name_wrapper_hw_platform_0/. Hit 'Add' again to add the file and set the partition type as datafile and click 'OK'.

- Now add u-boot file created earlier in the lab. Hit 'Add' again to add the file and select partition type as datafile.

- It is very important that the 3 files are added in this order, or else the FSBL will not work properly. It is also very important that you set FSBL.elf as the bootloader and system.bit and u-boot.elf as data files. Click Create Image and the created BIN file is named as BOOT.bin and you should be able to find it in the lab4 directory.

Linux Kernel

- Untar the file Linux kernel source code file 'linux-3.14.tar.gz' from a shared folder into your lab 4 directory.

- The Linux kernel can be configured to run on several devices. We need to configure the kernel with the default configuration for our board ZYBO. The configuration is located at arch/arm/configs/xilinx_zynq_defconfig.

- We need a cross compiler to build linux for ARM processor on Zynq. Navigate to linux source code folder 'linux-3.14' under the lab 4 directory and to use the default configuration type the following command.
  $ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- xilinx_zynq_defconfig

- Now that we have configured linux for our hardware we can compile it. Type the following command to compile linux.
  >make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-

- After successful linux compilation, you should see arch/arm/boot/zImage ready in the last few lines in terminal which is the kernel image.

- The kernel image is located at arch/arm/boot/zImage. However, in this case the kernel image has to be a uImage (unzipped) instead of a zimage. To convert image we need u-boot tools and we should include them in the PATH to use them. To include them in the path type the following

command.
>PATH=$PATH:<directory_to_u_boot>/tools

- The tools are ready and to make the uimage type the following command from Linux directory.
  >make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- UIMAGE_LOADADDR=0x8000 uImage.
- Once the image is converted you should see uImage in lab4/linux-3.14/arch/arm/boot/

- To boot the Linux operating system on the ZYBO board, you need BOOT.BIN, a Linux kernel image (uImage), a device tree blob (.dtb file), and a file system. BOOT.BIN and uImage have been created earlier in the lab. We will now compile the .dtb file. The default device tree source file is located in the Linux kernel source at arch/arm/boot/dts/zynq-zybo.dts.

- We have added a custom IP to the hardware system, hence we need to create an entry describing the 'multiply' IP in .dts file.

- 'multiply' IP is connected to the PS through an axi interconnect. Hence under 'ps7_axi_interconnect_0: amba@0' add the following lines after line 331.

```
multiply {
 compatible = "ecen449,multiply";
 reg = <0x43C00000 0x10000>;
};
```

- The reg entry holds the address of the multiply module which is the address allocated by Vivado in the 'Address Editor' tab to the multiply ip.

- Before we boot Linux we need to convert the .dts file to a .dtb(device tree binary) format which is the compatible format for arm. Execute the following command from the Linux source directory to convert the .dts file to .dtb.

- >./scripts/dtc/dtc -I dts -O dtb -o ./devicetree.dtb arch/arm/boot/dts/zynq-zybo.dts

- Copy the ramdisk file to the ECEN 449 directory. To wrap header file we need to use mkimage tools in u-boot directory. Run the following command from the ECEN 449 directory:
  >./u-boot/tools/mkimage -A arm -T ramdisk -c gzip -d ./ramdisk8M.image.gz uramdisk.image.gz

- The above command will create the uramdisk.image.gz file which we will use to boot linux

- Now we have all the files ready and its time to boot Linux on the ZYBO board. Copy the BOOT.bin, uImage, uramdisk.image.gz and devicetree.dtb file on to the SD card .

Simulation

Boot Linux on ZYBO

- Open a new terminal and source settings64.sh. We will use PICOCOM to watch the output from the ZYBO board.

- Change JP5 into SD card mode to boot from the SD card and power on the ZYBO board. Connect the USB cable to ZYBO board. Run the following command to start PICOCOM.

  $ picocom -b 115200 -r -l /dev/ttyUSB1

(To exit press Ctrl-A and then Ctrl-x to exit picocom.)

- Disconnect the SD card from the PC and plug it into the ZYBO board. Press the reset button(PSSRST) to start booting Linux. If the process is successful you should see Linux booting up on the ZYBO board via the PICOCOM console. Demonstrate this to the TA.

Evaluation of Results

This experiment focused on bringing up a Linux operating system on the ZYBO board by constructing a bootable image consisting of a hardware bitstream, FSBL, U-Boot, Linux kernel, and root filesystem. The lab involved building a custom hardware platform in Vivado that included peripherals necessary for Linux operation—such as SD card, UART, DDR3 RAM, and interrupt controllers. It also required the compilation of U-Boot and Linux using cross-compilation toolchains and the creation of a device tree that incorporated a previously designed hardware multiplier peripheral.

By completing this lab, we developed a comprehensive understanding of the embedded Linux boot process on ARM-based SoCs. We practiced configuring and compiling kernel and bootloader components, and integrating a hardware accelerator into a Linux-visible system via device tree modifications.

This workflow mirrors many real-life embedded systems, such as in IoT devices, routers, drones, and industrial controllers, where Linux offers both the flexibility of software and the performance of hardware acceleration.

- Compared to the previous lab, this lab's microprocessor system, shown in Figure 1, has 512 MB of SDRAM. However, our system still includes a small amount of local memory. What is the function of the local memory? Does this 'local memory" exist on a standard motherboard? If so, where?

- After your Linux system boots, navigate through the various directories. Determine which of these directories are writable. (Note that the man page for 'ls' may be helpful).

  Test the permissions by typing 'touch <filename>' in each of the directories. If the file, <filename>, is created, that directory is writable. Suppose you are able to create a file in one of these directories. What happens to this file when you restart the ZYBO board? Why?

- If you were to add another peripheral to your system after compiling the kernel, which of the above steps would you have to repeat? Why?

Safety Precautions

References

- https://digilent.com/reference/programmable-logic/guides/getting-started-with-vivado
- https://digilent.com/reference/programmable-logic/zybo-z7/reference-manual
- https://digilent.com/reference/programmable-logic/guides/installing-vivado-and-vitis
- https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM
- https://digilent.com/reference/programmable-logic/guides/getting-started-with-vivado