ELEC6234 – Embedded Processors

Coursework Report – picoMIPS implementation

Akash Biyani
ab7n23
Masters in Embedded Computing Systems
Tomasz Kazmierski

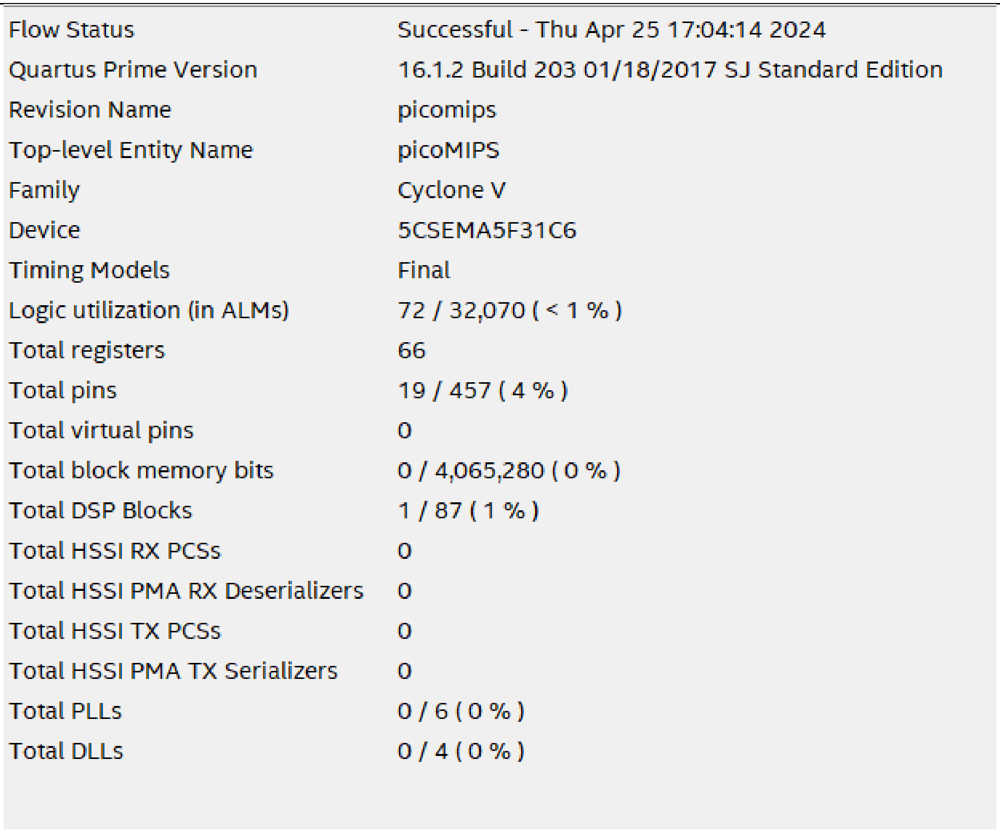
1. Introduction

The objective of this exercise is to design an 8-bit implementation of an application specific picoMIPS that implements the affine transformation algorithm of graphic pixels and implement on the Altera FPGA development board using smallest possible hardware.

The Affine transformation is successfully achieved. When provided 8-bit inputs X1 and Y1 (signed or unsigned), we receive correct outputs X2 and Y2 after the affine transformation algorithm. A customized instruction set is used to handle the flow of affine transformation algorithm which takes the inputs from switches of FPGA board and execute them as per the provided pseudocode.

There are 2 set of constants provided and first set of values are being used for affine transformation. Apart from basic picoMIPS structure an additional MUX is used to take input from the switch and out of this MUX is provided to ALU MUX. If the switch MUX is enabled and ALU MUX is enabled then input from switch is provided to ALU input 2 otherwise Immediate literal (constant set of values for affine transformation) are provided to ALU input 2. The data stored in register is sent to ALU input 1 at the same time to perform the operation based on instruction set. Instruction named ADDF is used to store the input from the switch. There are 2 branching instruction for relative and absolute branching. Branching is executed when connect to SW[8]) ==Instruction [7], and this is also known as the handshake function.

Design Details Form			
Total Cost: 72	**ALMs: 72**	**Memory bits: 0**	**Multipliers: 1**
Flow Summary			



Flow Status	Successful - Thu Apr 25 17:04:14 2024
Quartus Prime Version	16.1.2 Build 203 01/18/2017 SJ Standard Edition
Revision Name	picomips
Top-level Entity Name	picoMIPS
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	72 / 32,070 (< 1 %)
Total registers	66
Total pins	19 / 457 (4 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	1 / 87 (1 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Fig 1: Resource Utilization

Instruction Set

A total of 9 instructions have been implemented to ensure that basic operations are satisfied.
Details:
ADD : Adds the values stored in the source and destination registers and stores the result in the destination register.
ADDI : Add the immediate value to the value stored in the source register and store the result in the destination register.
ADDF: Add the immediate value fetched from the input (switches) and the value stored in the source register, and store the result in the destination register.
MUL : Multiplies the values stored in the source and destination registers and stores the result in the destination register.
MULI : Multiplies the immediate value to the value stored in the source register and store the result in the destination register.
BREL: Branch at a specified condition that is sw[8] == instruction[7]
BABS: Branch at the end of execution for next execution and jump to specific instruction
NOP : No Operation

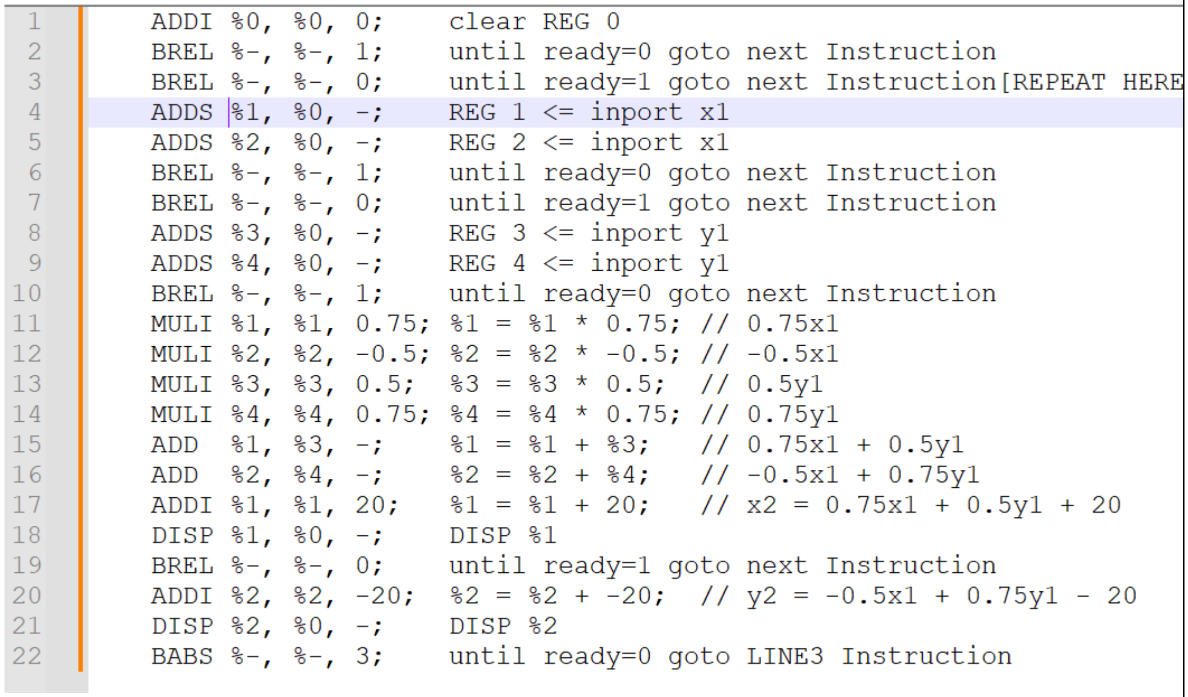
Instruction Format

Data-bus = 8bit, Instruction size = 24 bits,
Format of instruction: 6-bit opcode, 5-bit address destination register, 5-bit address for source register, 8-bit data for immediate literals

ADD	: ADD %d, %s;	%d = %d + %s
ADDI	: ADDI %d, %s, imm ;	%d = %s + imm
ADDF	: ADDF %d, 0;	input immediate value
MUL	: MUL %d, %s;	%d = %d * %s

MULI	: MULI %d, %s,	imm ; %d = %s * imm
BREL	: BREL Address,	Branch at specified condition sw[8] == instruction[7]
BABS	: BABS Address,	Jump to specified instruction
NOP	: No operation,	No operation

Your affine transformation program



```
 1   ADDI %0, %0, 0;     clear REG 0
 2   BREL %-, %-, 1;     until ready=0 goto next Instruction
 3   BREL %-, %-, 0;     until ready=1 goto next Instruction[REPEAT HERE
 4   ADDS %1, %0, -;     REG 1 <= inport x1
 5   ADDS %2, %0, -;     REG 2 <= inport x1
 6   BREL %-, %-, 1;     until ready=0 goto next Instruction
 7   BREL %-, %-, 0;     until ready=1 goto next Instruction
 8   ADDS %3, %0, -;     REG 3 <= inport y1
 9   ADDS %4, %0, -;     REG 4 <= inport y1
10   BREL %-, %-, 1;     until ready=0 goto next Instruction
11   MULI %1, %1, 0.75;  %1 = %1 * 0.75; // 0.75x1
12   MULI %2, %2, -0.5;  %2 = %2 * -0.5; // -0.5x1
13   MULI %3, %3, 0.5;   %3 = %3 * 0.5;  // 0.5y1
14   MULI %4, %4, 0.75;  %4 = %4 * 0.75; // 0.75y1
15   ADD  %1, %3, -;     %1 = %1 + %3;   // 0.75x1 + 0.5y1
16   ADD  %2, %4, -;     %2 = %2 + %4;   // -0.5x1 + 0.75y1
17   ADDI %1, %1, 20;    %1 = %1 + 20;   // x2 = 0.75x1 + 0.5y1 + 20
18   DISP %1, %0, -;     DISP %1
19   BREL %-, %-, 0;     until ready=1 goto next Instruction
20   ADDI %2, %2, -20;   %2 = %2 + -20;  // y2 = -0.5x1 + 0.75y1 - 20
21   DISP %2, %0, -;     DISP %2
22   BABS %-, %-, 3;     until ready=0 goto LINE3 Instruction
```

Fig 2: Affine transformation algorithm

Design Block Diagram showing your modules, data signals and control signals

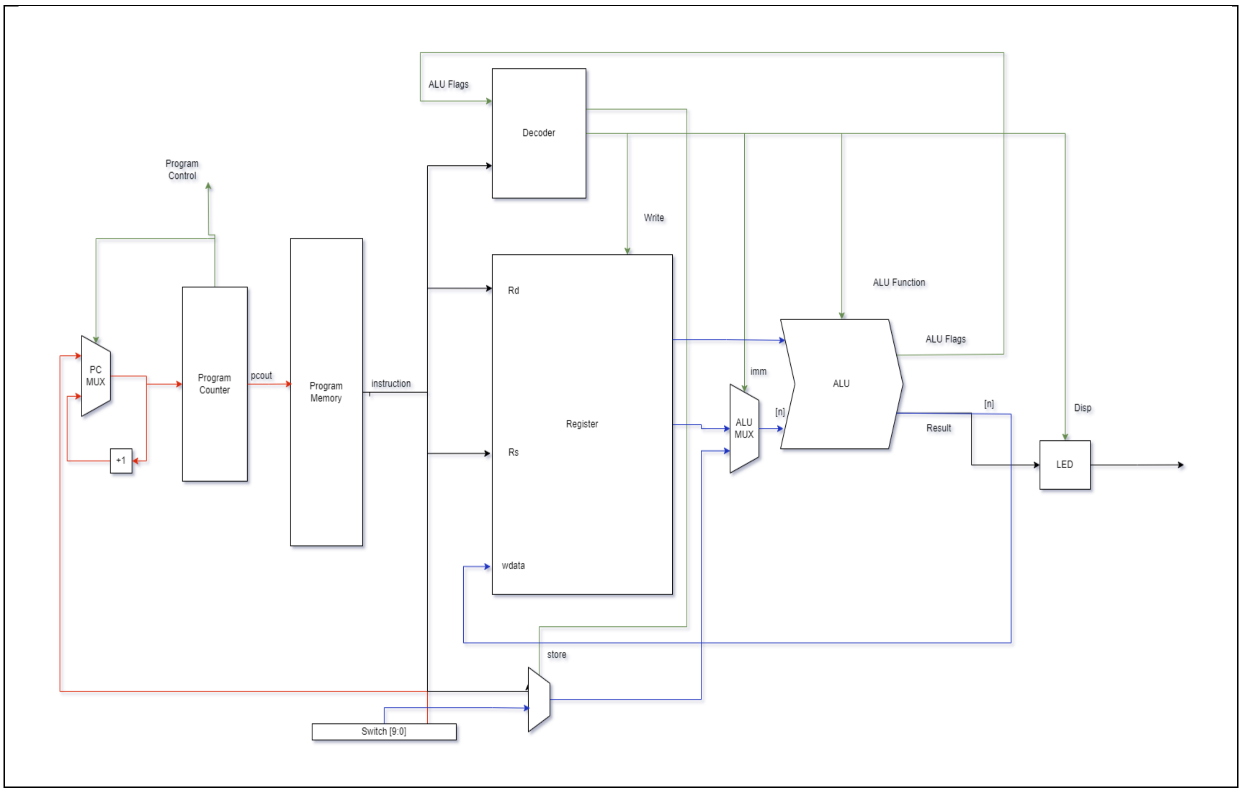


Fig 3: Design Block diagram (Green – control path, Red – Address, Blue – Data path)

2. Overall architecture of the design and simulations

In the architectural design I've devised, the framework integrates an arithmetic logic unit (ALU), a general-purpose register (GPR), a 24-bit instruction decoder, a program counter (PC), and a program memory housing the machine code for the affine transformation algorithm. These components are interconnected according to the depicted design block diagram above. The input switches conveying coordinate data are linked to the ALU block through multiplexers.

The affine transformation can be simplified and expressed by the equation shown below.
$X2 = (A11 \times X1 + A12 \times Y1) + B1$
$Y2 = (A21 \times X1 + A22 \times Y1) + B2$

From the above equation, we understand that we require 4 multiplication and 4 addition operations. We use a 8 bits data bus width for the picoMIPS and 24 bits instruction. For the implementation of the multiplication operation, we can use the embedded multipliers in the dsp block. The picoMIPS architecture is more than enough to implement the affine transformation.

Following image shows simulation of picoMIPS module. For the provided input accurate output value of X2 and Y2 is received when ready is 0 and 1 respectively. This concludes that picoMIPS is functioning as per designed.

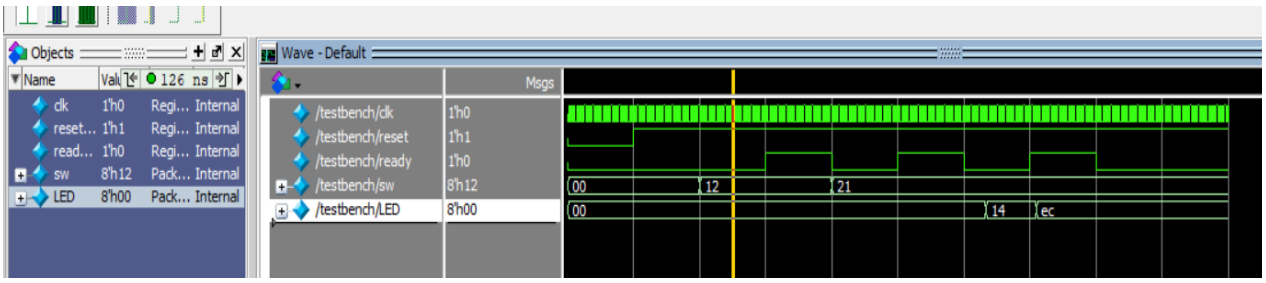


Fig 4: picoMIPS simulation

Simulation of all the individual modules separately is carried and a testbench is created for each module to test if the inputs and outputs are as required.

Program memory: For every provided address, program memory outputs the instruction. From processors perspective address is received from program counter and instructions are sent to different modules which are specific parts of the entire 24-bit instruction.

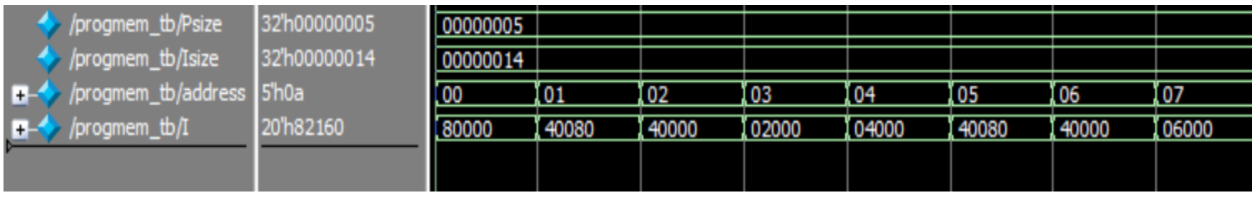


Fig 5: Program memory simulation

GPR: There are 8 general purpose registers of 32-bit created, only 4 of these are currently used. Reg %1 and %2 to store X1, reg %2 and %3 to store Y1. The calculations are carried out and result is saved in these registers. As shown below, system is reset when active low. On positive clock, next_gpr is assigned to gpr.

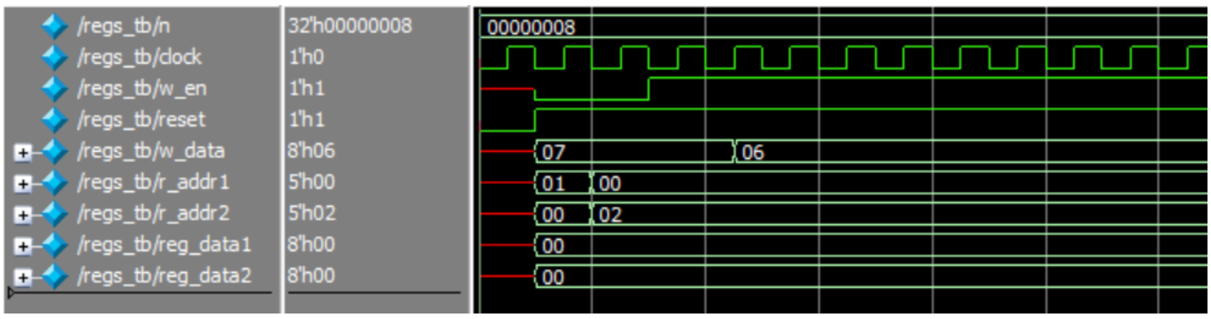
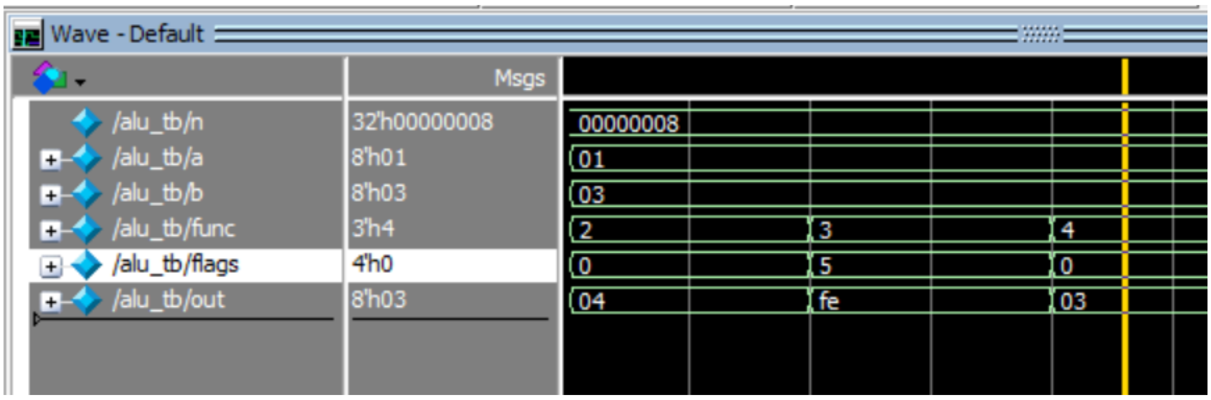


Fig 6: GPR simulation

Decoder: Decoder receives 6-bit opcodes from program memory and instruction set defined in hex file and based on that it sets the ALU functionality, the enable signals for MUX to select the immediate literal or the register data for operation. When simulated individually decoder sets proper ALU functions based on input instruction set.

ALU: ALU is performs all the operation and it is necessary to ensure the 2's complement of input is executed if subtraction is to be carried out which is taken care of in first always comb block. MY ALU is performing Addition, Subtraction, Multiplication and setting the flags as well. The result of two 8-bit input multiplication is truncated to [14:7] bits as explained in coursework instruction. K-map is drawn for carry and overflow and required flags are set as per that. I have tested ALU implementation separately again to ensure the properly. For input 1 and 3 addition, subtraction and multiplication is carried out and is correct (4, -2, 3)



```
Wave - Default
                    Msgs
/alu_tb/n           32'h00000008   00000008
/alu_tb/a           8'h01          01
/alu_tb/b           8'h03          03
/alu_tb/func        3'h4           2         3         4
/alu_tb/flags       4'h0           0         5         0
/alu_tb/out         8'h03          04        fe        03
```

Fig 7: ALU simulation

Testbench for simulation of all the modules are included in zip file.

3. FPGA implementation

For implementation on FPGA, step one was to synthesize the code in Quartus Prime. I had to go through a few errors as I had not initialised some parameters. After nullifying the errors, the FPGA synthesis was successful. As per the coursework instruction, I used the clock divider to slow down the clock otherwise it would be too fast for manual inputs. The input vectors were provided via switches and results were checked as per the provided pseudocode.

We are required to design with minimum hardware to avoid additional cost, therefore, we have to change the optimisation technique from “Balanced” to “Area” for synthesis in Quartus Prime. This in turn reduces the resource cost.

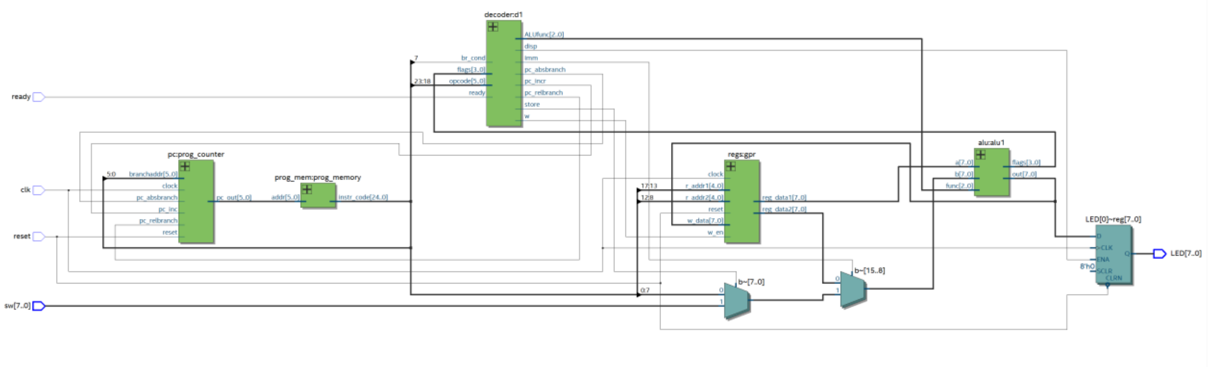


Fig 8: picoMIPS RTL Level Diagram

Above is the RTL level diagram generated by Quartus Prime, this seems to be matching with the Design block diagram. The control path and data path are visible clearly. Apart from that, modules such as Program Counter, Program Memory, Decoder, Register, ALU, which contains main CPU parts are also visible. A more clearer RTL diagram pdf file is attached in the zip file along with code.

4. Conclusion

The objective of the project to design an 8-bit implementation of an application specific picoMIPS that implements the affine transformation algorithm of graphic pixels and implement on the Altera FPGA development was successfully achieved. To reduce the cost of design I tried to reduce instruction size from 24-bit to 20-bit (reducing address size from 5-bit to 3-bit for each register). However, this did not affect the resultant cost.
The processor is successfully implemented, simulated and synthesized on Altera FPGA board for affine transformation of pixel graphics.

This project has helped to boost the understanding of processor and build the confidence of implementing embedded processor for small tasks.

1. 7. References

1] Dr Tom J Kazmierski, “ELEC6234 Embedded Processor Synthesis: Notes,” University of Southampton [Online]. Available: https://secure.ecs.soton.ac.uk/notes/elec6234/
[2] ZwolińskiM., Digital system design with SystemVerilog. Upper Saddle River, Nj: Addison-Wesley, 2010.