ELEC6234 – Embedded Processors
Coursework Report – picoMIPS implementation

Ahmed Soliman
As8u22
EMECS – Embedded Computing Systems – Part 1
Tomasz Kazmierski

1. Introduction

This report presents the design and simulation of a processor architecture used for implementing an affine transformation algorithm on 2 input and gives output values afterwards. It is inspired by the PicoRisc Embedded architecture design. This was done by studying the design architecture through the lecture slides and understanding the file examples given for each module. Then each module was created and tested through simulation to verify its operation. Then integration of the modules was made under an encapsulating module (CPU). Modifications such as changing instruction format and creating unique instructions were made to reduce the size of the design as much as possible while keeping it efficient and fast. Finally, design was uploaded on FPGA to synthesize it which showed correct behaviour in real life similar to the simulation that was made on Modelsim.

Design Details Form

Total Cost: 50	**ALMs: 50**	**Memory bits: 0**	**Multipliers: 1**

Instruction Set

ADDI	3'b001	//Add register with number directly
ADD	3'b010	//Add 2 registers
BNE	3'b011	//branch if registers are not equal
BEQ	3'b100	//branch if registers are equal
LDI	3'b101	//Put value of one register to the other
MUL	3'b110	//Multiply 2 registers

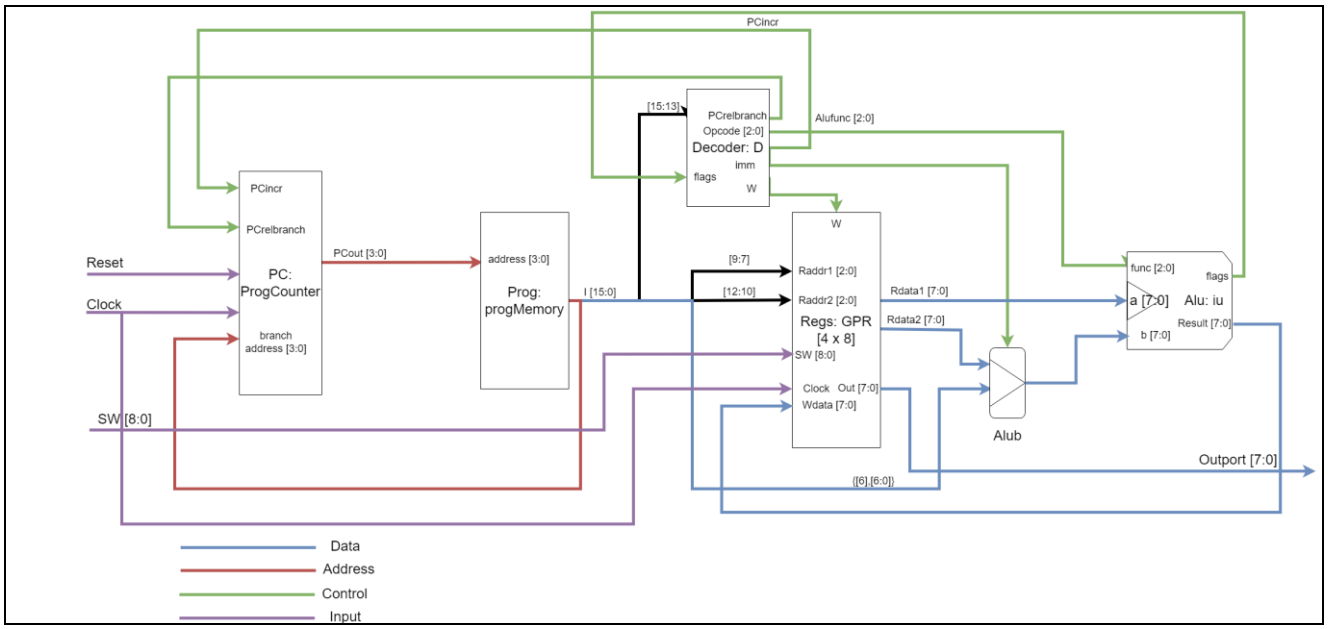
Instruction Format

format: 3b opcode, 3b %d, 3b %s, 7b immediate or address
%d: is passed to Raddr2 in reg.sv
%s: is passed to Raddr1 in reg.sv
Instruction Size is 16 bits
e.g. 4E80 // 16'b010_011_101_000_0000 ADD %3, %3, %5; %3=%3+%5

Your affine transformation program

```
// HEX ////////// BINARY ////////////////////// ASSEMBLER /////////
B000  // 16'b101_100_000_000_0000  LOAD %4, %4, %0; %4 = %0, load 0 in reg 4
8080  // 16'b100_000_001_000_0000  BEQ %0, %1, -1; branch if %1=0
CD30  // 16'b110_011_010_011_0000  MUL %3, %2, 0.75; %3=%2*0.75
D960  // 16'b110_110_010_110_0000  MUL %6, %2, -0.5; %6=%2*-0.5
6080  // 16'b011_000_001_000_0000  BNE %0, %1, -1; branch if %1!=0
8080  // 16'b100_000_001_000_0000  BEQ %0, %1, -1; branch if %1=0
D520  // 16'b110_101_010_010_0000  MUL %5, %2, 0.5; %5=%2*0.5
4E80  // 16'b010_011_101_000_0000  ADD %3, %3, %5; %3=%3+%5
D530  // 16'b110_101_010_011_0000  MUL %5, %2, 0.75; %5=%2*0.75
6080  // 16'b011_000_001_000_0000  BNE %0, %1, -1; branch if %1!=0
3194  // 16'b001_100_011_001_0100  ADDI %4, %3, 20; %4=%3+20
5A80  // 16'b010_110_101_000_0000  ADD %6, %6, %5; %6=%6+%5
8080  // 16'b100_000_001_000_0000  BEQ %0, %1, -1; branch if %1=0
336C  // 16'b001_100_110_110_1100  ADDI %4, %6, -20; %4=%6-20
6080  // 16'b011_000_001_000_0000  BNE %0, %1, -1; branch if %1!=0
80F1  // 16'b100_000_001_111_0001  BEQ %0, %1, -15; branch if %1=0
```

Design Block Diagram showing your modules, data signals and control signals (do not use Quartus generated RTL diagrams)



2. Overall architecture of the design and simulations

My processor architecture consists of a pc, Prog memory, Regs [4 x 8 bits], Decoder, Alu. A top-level module called CPU encapsulate these blocks in one module.
The **CPU** module as said is a top-level module that instantiate the blocks of the processor while creating the signals needed for each module to operate. A **MUX** is created in the CPU to choose between second register of the Regs module and the immediate value from the prog instruction to be passed to the Alu module.

The **PC** module has inputs of clock, reset, PCincr, PCrelbranch (flags), and branchaddress, and has an output of PCout. The default value for PC size is 6 bits which can carry up to 64 instructions, however in our application it was set to 4 bits as we only need 16 instructions for our code. Basically, the PC operation is by incrementing its value each clock cycle to be able to move to next instruction of the code with an additional feature of branching to an address if PCrelbranch was set by adding the relative branch to the current PC value. Using absolute branch address in the PC was not needed which is why it was removed from it. As seen in Figure 1, PC either increment or keep its value (branch with relative value 0) depending on the Input of the Switch with the Instruction of the memory code

In the **ProgMemory**, the module creates a memory block to store the instructions. Then the PC is given as an input address and the output of the module is the instruction code for this address. For our application the memory needed was only 16 instructions of size 16 bits, this reduction was done to reduce the cost size. Showing in Figure 1, Instruction code value depend on PC value.

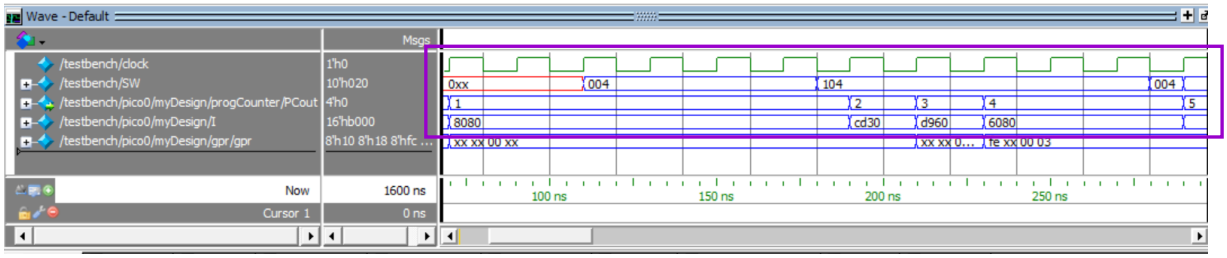


Figure 1. PC incrementing value and instruction change accordingly

The **Decoder** module is needed to take decision based on opcode given (3 bits) and the Alu flag (1 bit) decides whether it is an ADD, ADDI, BEQ, BNE, LDI, MUL. Then it gives output func for the Alu, output flag w for register, output flag imm for MUX, and output flags PCincr, PCrelbranch for PC. All of these signals are control signals for the modules. Opcode was reduced from 6 bits to just 3 bits as operations needed in our code were less than 8 instructions. In Figure 2 we can see the change of values for Alufunc, imm, w depending on the decision decoder took from the input of the opcode.

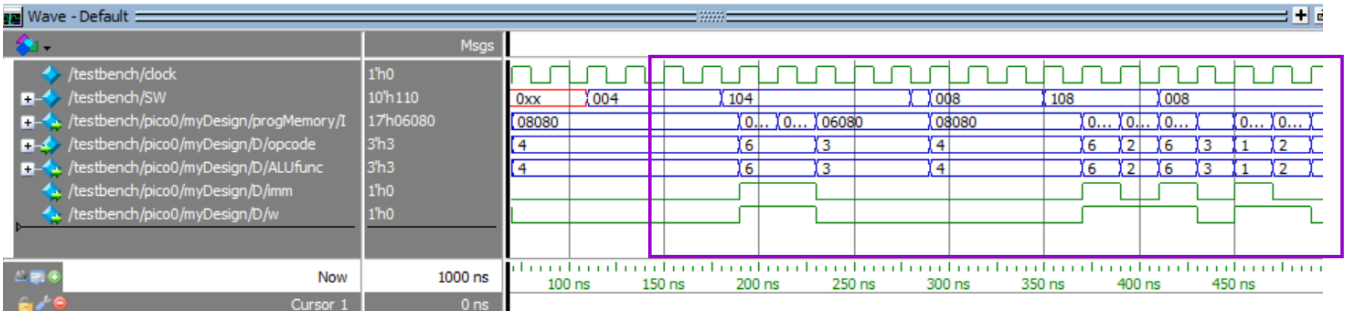


Figure 2. Decoder Parameters changing according to opcode given

For the **Regs** module, it takes inputs of clock, w, Raddr1, Raddr2, SW. ProgMemory sends addresses of the 2 registers we need to use and based on w flag we write on the second register or not. Reg %0 is reserved for 0 value, Reg %1 is reserved for handshake signal from the output switch, Reg %2 is reserved for input data of the switch. The rest of the Regs are for internal calculations operations %2 ➔ %6. In reality, we only need a GPR of size 4 Regs as %0 ➔ %2 is either direct value or outside value from switches. This helps in reducing the size of GPR from initial 32 Regs to just 4 Regs. Outputs of this module are the data of the 2 Regs accessed with the addresses, and the output LED of the FPGA is connected to Regs %4 ➔ gpr[1]. For example, in Figure 3, we can see in the violet box an operation is made on addresses %4, %3 ➔ gpr[1] , gpr[0] which also reflect on **out** variable.

```
// Declare 4 n-bit registers
logic [n-1:0] gpr [3:0];
assign out =gpr[1];
// write process, dest reg is Raddr2
always_ff @ (posedge clk)
begin
        if (w)
    gpr[Raddr2-3] <= Wdata;

end
```

```
// read process, output 0 if %0 is selected, output SW[8] if %1,
// output SW[7:0] if %2
always_comb
begin
   if (Raddr1==3'd0)
         Rdata1 =  {n{1'b0}};
else if (Raddr1==3'b001)
        Rdata1={7'b0000000,SW[8]};
else if (Raddr1==3'b010)
        Rdata1=SW[7:0];
else  Rdata1 = gpr[Raddr1-3];

if (Raddr2==3'd0)
         Rdata2 =  {n{1'b0}};
else if (Raddr2==3'b010)
        Rdata2=SW[7:0];
  else  Rdata2 = gpr[Raddr2-3];
end
```

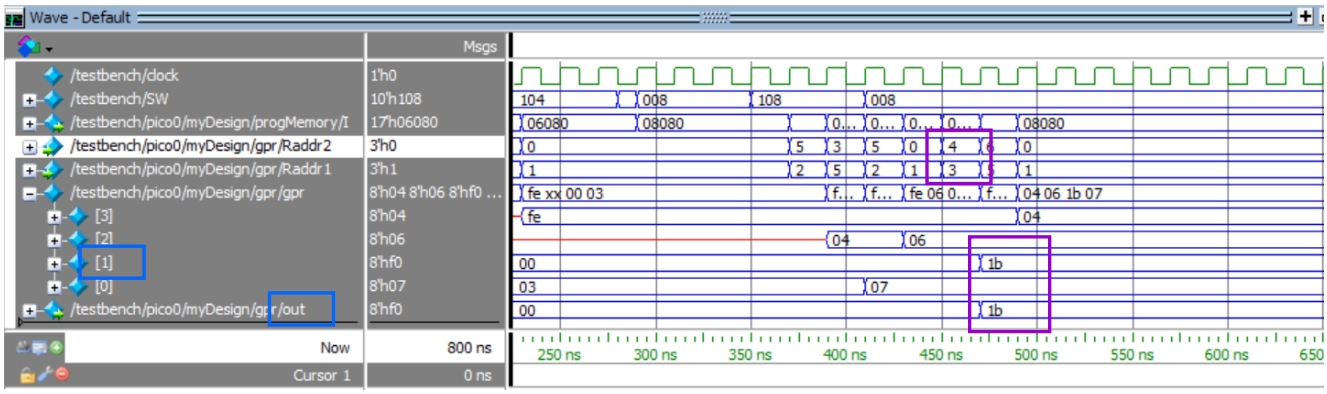


Figure 3. Register values get stored according to address given and out change with register 1

Finally, **Alu** takes input data a, b, and the func control signal to determine the operation (add, Sub, Mult, Load). For subtraction we just get 2's comp for b and then use addition operation. For multiplication an instantiation of the multiplier of the DSP block is created and we take only the [14:7] bits of the multiplication result to the output of the Alu. Alu's output is the result data of the operation done and the flag control signal for the decoder. In our code we only need one flag (Z) which checks if the result is 0 or not. In Figure 4, we can see that depending on the func an operation is done and result is calculated and if w is 1 the result will be stored in one of the GPR registers. Also, flag value is calculated to be passed to the decoder

```
signed_mult m(.out(Rmul), .a(a), .b({b[n-2:0],1'b0}));
// calculate flags Z
flag = result == {n{1'b0}}; // Z
`RMUL  : result = Rmul[14:7];
endcase
```

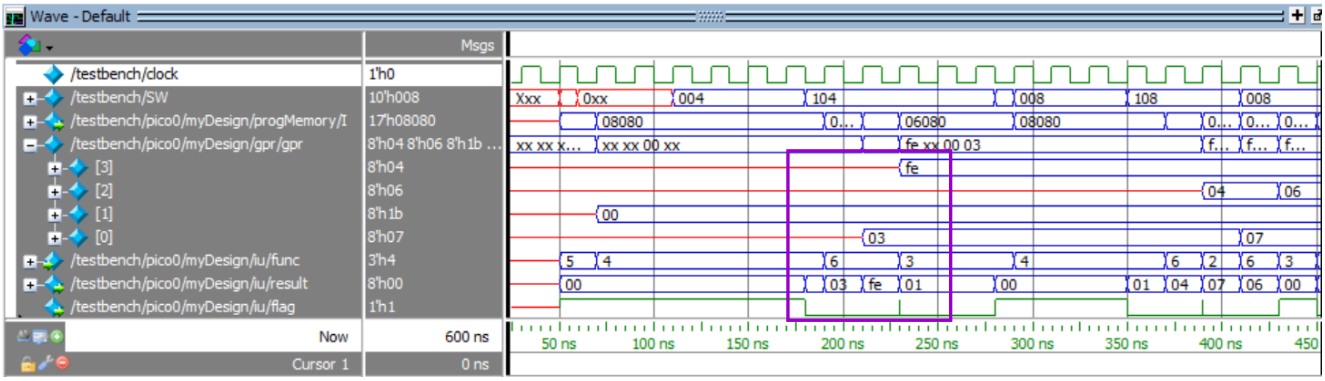


Figure 4. Alu result and flag is calculated and result is stored in one of the registers

Finally for a full model simulation, Inputs of values 4, 8 were given with a specific matrix A {0.75, 0.5, -0.5, 0.75}, C {20, -20} and expected output of 27 (hex: 1B), -16 (hex: F0) should be calculated. And as seen in Figure 8, these outputs are shown in **out** variable & gpr[1] then **PC** returns to zero and program start over again.

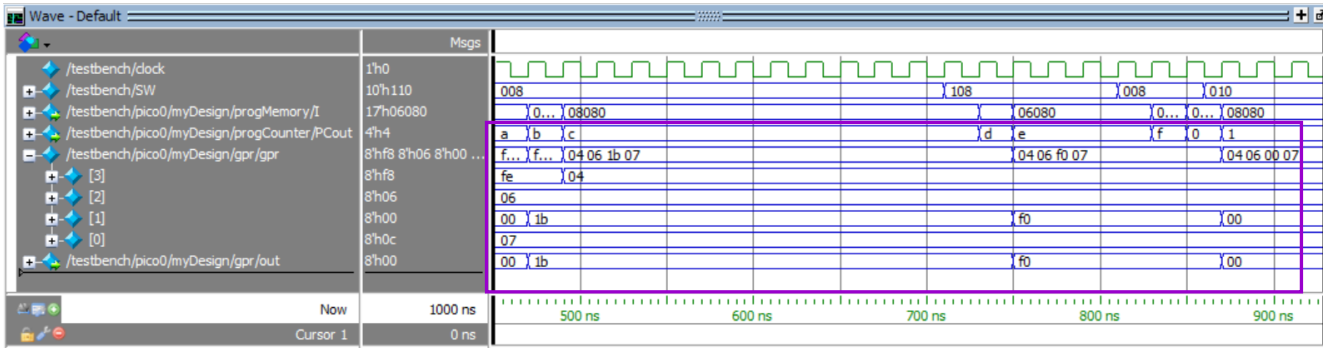


Figure 5. Final result is shown in out variable and pc return to 0 again

3. FPGA implementation

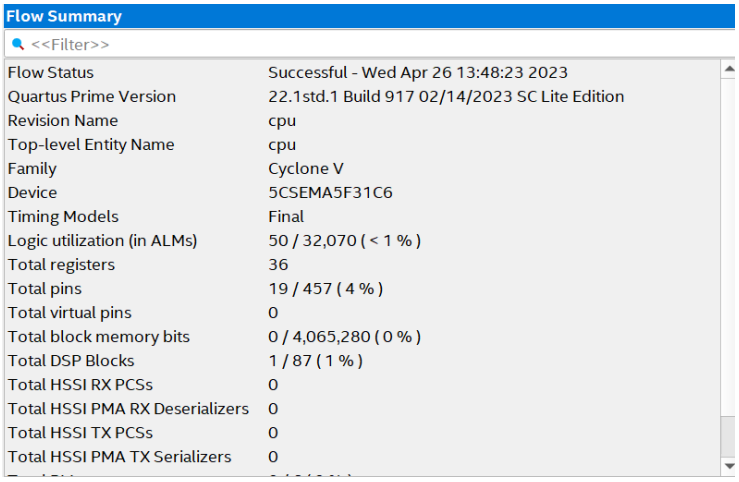
After Finishing the processor design and simulating it on Modelsim using a testbench that I created. I used Quartus to synthesize the code and upload it on the FPGA. The code worked perfectly on the FPGA with same behaviour as the simulation in addition to other tests however the size of it was much larger than needed. Initially I had 118 ALMs so I had to redesign my code to make it more efficient. Firstly, I removed all unnecessary instructions that were not used like BGE, BLO, RAND and others. And removed their implementations from the decoder and alu modules. Next, I reduced the registers array from 32 to only 4. Also, I reduced the instruction size from 24 bit to 16 by reducing opcodes to 3 bits, register address to 3 bits, immediate value to 7 bits.

```
5A80    // 16'b010_110_101_000_0000 ADD %6, %6, %5;  %6=%6+%5
```

And to preserve the value of numbers for the immediate after reducing to 7 bits I used the MSB (bit 6) to be used again as bit 7 in adding operations while in multiplying as it is a fraction so I removed this MSB added in Alub.

```
// create MUX for immediate operand
assign Alub = (imm ? {I[n-2],I[n-2:0]} : Rdata2);
signed_mult m(.out(Rmul), .a(a), .b({b[n-2:0],1'b0}));
```

As for the program code, several iterations were made to optimize it as it went from 26 instructions to just 16. This was done through better utilizations of instructions and removing unnecessary ones like NOP while changing in the algorithm for calculating the affine transformation for optimizing the usage of registers. This is done by doing calculations as soon as first input is present instead of waiting for both inputs and storing them in registers then do calculations which wastes registers in just storing values. All of that contributed to reducing the cost size to just 50 ALMs with no memory bits.



Flow Summary	
<<Filter>>	
Flow Status	Successful - Wed Apr 26 13:48:23 2023
Quartus Prime Version	22.1std.1 Build 917 02/14/2023 SC Lite Edition
Revision Name	cpu
Top-level Entity Name	cpu
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	50 / 32,070 (< 1 %)
Total registers	36
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

Figure 6. Flow summary of Design

Next, I needed to assign FPGA Pins to the qsf file to be able to control the inputs and display the outputs on the LEDs, so I had to open the FPGA pin configuration pdf to know the values of the specific pins needed which can be seen in Figure 7.

```
71  set_location_assignment PIN_AB12 -to SW[0]
72  set_location_assignment PIN_AC12 -to SW[1]
73  set_location_assignment PIN_AF9 -to SW[2]
74  set_location_assignment PIN_AF10 -to SW[3]
75  set_location_assignment PIN_AD11 -to SW[4]
76  set_location_assignment PIN_AD12 -to SW[5]
77  set_location_assignment PIN_AE11 -to SW[6]
78  set_location_assignment PIN_AC9 -to SW[7]
79  set_location_assignment PIN_AE12 -to SW[9]
80  set_location_assignment PIN_AF14 -to fastclk
81  set_location_assignment PIN_V16 -to LED[0]
82  set_location_assignment PIN_W16 -to LED[1]
83  set_location_assignment PIN_V17 -to LED[2]
84  set_location_assignment PIN_V18 -to LED[3]
85  set_location_assignment PIN_W17 -to LED[4]
86  set_location_assignment PIN_W19 -to LED[5]
87  set_location_assignment PIN_Y19 -to LED[6]
88  set_location_assignment PIN_W20 -to LED[7]
89  set_location_assignment PIN_AD10 -to SW[8]
```

Figure 7. pins allocation to variables

I tried using the fastclock of the FPGA (50 MHz) but it showed strange behaviour and didn't work properly this is due to the bouncing of the switches in real time comparing to the fast clock that the programming was working on. So, I realised that it is best to use the slow clock counter provided which made the code work properly and without any strange behaviour.

4. Conclusion

In conclusion, the successful implementation of an affine transformation algorithm on 2 input values was presented in this report, achieved through the design and simulation of a processor architecture inspired by the PicoRisc Embedded architecture. During the course of this project, I gained knowledge on various aspects such as analysing an Embedded design structure, generating and evaluating individual modules, merging the modules into a cohesive encapsulated module, and adapting the instruction format by devising customized instructions to optimize the design's performance

To improve the design, we could add more modules to increase the functionality of the processor, such as a memory module to store data, an input/output module to interface with external devices, and a stack module to handle function calls and returns. We could also optimize the design further by reducing the number of instructions needed and using more efficient algorithms for the operations. Overall, this project provided a valuable learning experience in embedded processor design and implementation.

1. 7. References

Zwolinski, M. (2017). Synthesis Walk-Through – Quartus and SystemVerilog [PDF file]. Retrieved from https://secure.ecs.soton.ac.uk/notes/adv_hdl/2017/QuartusWalkthrough.pdf

(2014). DE1-SoC User Manual. Imperial College London. Retrieved from http://www.ee.ic.ac.uk/pcheung/teaching/ee2_digital/de1-soc_user_manual.pdf