ELEC6234 – Embedded Processors

Coursework Report – picoMIPS implementation

Nahom Gebremichael
Ng3u24@soton.ac.uk
European Masters in Embedded computer systems
Dr Basel Halak

1. 1. Introduction

The instruction size has been reduced to 20 bits, and the overall system operates at super high speeds of 50 MHz. Since the actual 50 MHz clock is used, a debouncing logic is implemented, which samples the handshaking switch for 20 ms, meaning it counts for 500,000 cycles at the given 50 MHz clock. To fulfill the timing requirements, a system design constraint file(.sdc) file is written using timing analyzer. The hold and setup slacks are made positive, and unconstrained paths are constrained.

Custom load instructions for the handshaking signal and for the index switches are designed, and the required opcode and ALU code are assigned to each. Custom display commands are added to the decoder and ALU sections to assist in decoding; these display commands, in addition to the designed testbench, help debug the overall code. To ease tracking of the overall read-write operations, the program counter needs to be monitored, and the registers are checked when written to and read from.This custom-made handshaking signal acquisition technique enables the BEQ instruction to check the register and determine if the handshaking signal is asserted or not.

To decrease the size of the ROM, only the 256 noisy samples are stored in the ROM, and the kernels are stored in the program instructions as a literal values. To reduce the number of registers used, a chained addition technique is also implemented.

Design Details Form			
Total Cost: 248	**ALMs:** 248	**Memory bits:** 0	**Multipliers:** 1
- **ADD**: Fetches two values, one from source register1 and the other from source register2, adds them, and stores them in the destination register. - **MUL**: Fetches the value from the source register, multiplies it with the value in the immediate field, then stores the result in the destination register; the immediate field in this case stores the kernel values. - **LDSW**: Stores the debounced value of the handshaking signal to the destination register. - **LDI**: Loads a value from the source register to the destination register. - **LDROM**: Fetches the values stored in the rom based on the base value and the offset value and then stores it in the destination register. - **MOVSW**: Stores the values from the switches, i.e., the index value, to the destination register. - **LDRIND**: Fetches the values stored in the rom based on the base value and the offset			

value and then stores it in the destination register; when the offset value is zero, the instruction fetches the address from the same which is the base address; when the offset is one, it increments or decrements the base value and fetches a value from rom based on the new address and stores it in the destination register.
- **JMP**: Is an absolute branch where it branches to the desired address assigned on the intermediate field.
- **BEQ**: Is a conditional branch that jumps to the desired address, stored on the intermediate fields based on a fulfilled condition.

Instruction Format

R-Type (ADD):

Opcode [19:16]_	**Destination Register [15:11]_**	**Source Register 1 [10:6]**	**Source Register 2 [5:0]**

The instruction has 20 bits and the MSB bis from 16 to 19 are the opcodes, where the decode uses it to decide what type of operation it is , then the bits from 11 to 15 are the destination registers value, where the result is stored, and the bits from 6 to 10 are the source register 1 address and the bits from 0 to 5 are the source register2 address.

I-Type (MUL, LDI, LDRIND):

Opcode [19:16]_	Destination Register [15:11]_	Source Register 1 [10:6]	Immediate [5:0]

From the 20 bit instruction set, the bits from 16 to 19 are the opcodes which are decoded by t Decoder, the bits from 11 to 15 are the destination register, where the result is written , the bits from 6 to 10 are the source registers where the first operand is stored in case of the MUL, Where the base address in case of the LDRIND, and the bits from 0 to 5 are the intermediate values which are used to store the kernel values in case of the MUL instruction and the offset value in case of the LDRIND instruction

Load Type (LDSW, MOVSW):

Opcode [19:16]_	Destination register [15:11]_	**Unused [10:6]**	**Unused [5:0]**

Just like all the instruction sets. the bits from 16 to 19 is used as an opcode , then the bits from 11 to 15 are used as a destination register, to store the value, the debounced handshaking value in case of the LDSW and the base address (the index value from the switches).The bits from 0 to 5 are unused and opened for future additions.

Branch Type (JMP, BEQ):

Opcode [19 :16]_	**Destination register [15:11]_**	**Unused [10:6]**	**Address [5:0]**

Just like all the instructions, the bits from 16 to 19 are used an opcode and the bits from 11 to 15 are used as the destination register the bit from 6 to10 is unused and is opened for futur Additions. The bits from 0 to 5 are used as the address bits where the JMP or BEQ goes to

Your program

```
progMem[0] = 20'h00000;          // NOP
progMem[1] = 20'h42800;          // LDSW %5
progMem[2] = 20'hA2810;          // BEQ %5, %0, +16  (If %5 = 0, skip 16 instructions)
progMem[3] = 20'h75800;          // MOVSW %11, sw[6:0] (base address)
progMem[4] = 20'h832FE;          // LDRIND %6, rom[%11 -2 ]
progMem[5] = 20'h83AFF;          // LDRIND %7, rom[%11 -1
progMem[6] = 20'h842C0;          // LDRIND %8, rom[%11 + 0 ]
progMem[7] = 20'h84AC1;          // LDRIND %9, rom[%11 + 1 ]
progMem[8] = 20'h852C2;          // LDRIND %10, rom[%11 + 2 ]
progMem[9] = 20'h36191;          //%12 %6 *K[0]
progMem[10] = 20'h369DD;         //%13 %7 *K[0]
progMem[11] = 20'h37223;         //%14 %8 *K[0]
progMem[12] = 20'h37A5D;         //%15 %9 *K[0]
progMem[13] = 20'h38291;         //%16 %10 *K[0]
progMem[14] = 20'h20B0D;          //ADD %1, %12, %13
progMem[15] = 20'h20B81;          //ADD %1, %14, %1
progMem[16] = 20'h20BC1;          //ADD %1, %15, %1
progMem[17] = 20'h20C01;         //ADD %1, %16, %1
progMem[18] = 20'h51000;         // LDI %2, 0x00  (Load 0x00 into %2 if handshaking = 0)
progMem[19] = 20'h90000;         // JMP 0  (Loop back)
```

2. Overall architecture of the design and simulations

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

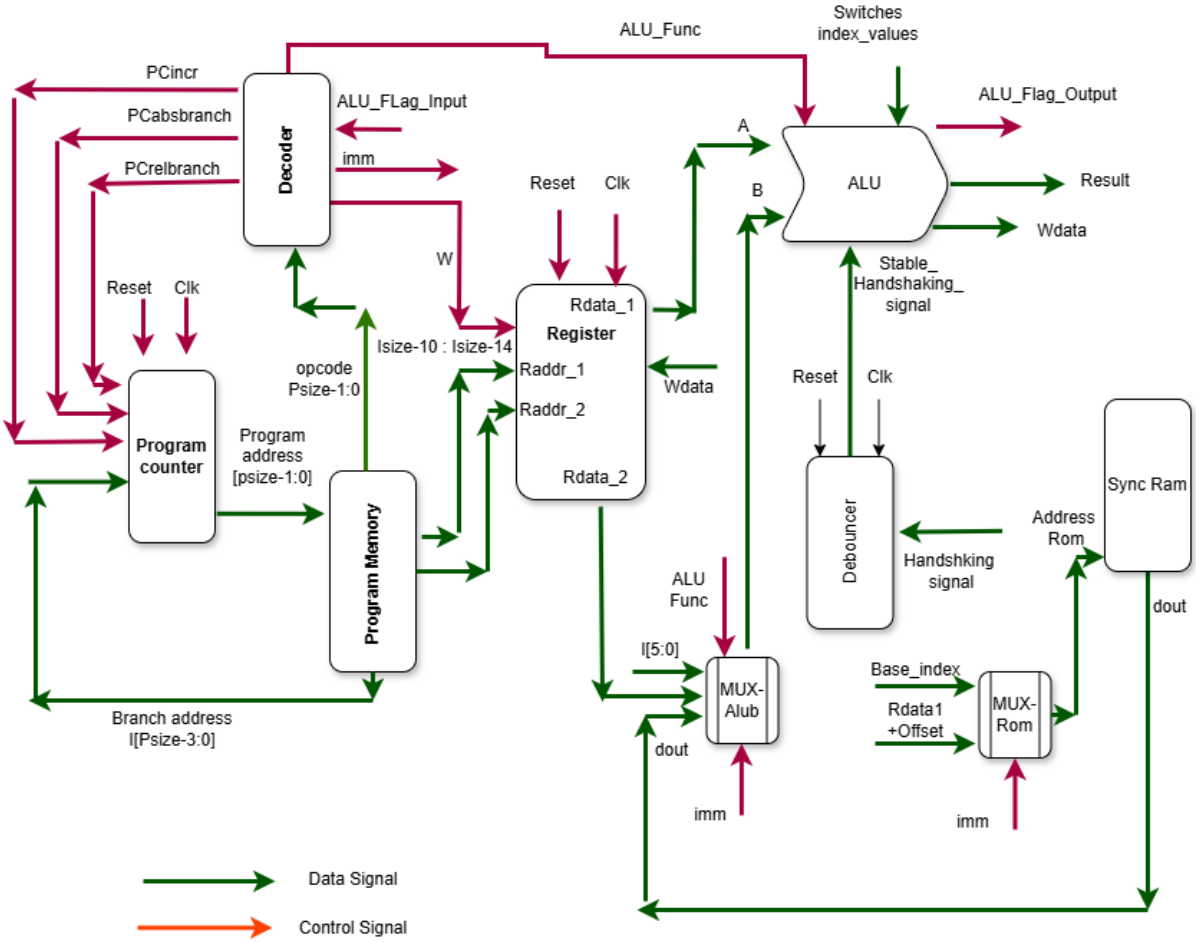

Figure 1.1 Control and Data signals of the processor

The picoMIPS processor comprises the following modules the Program Counter (pc) for generating an 8-bit instruction address (PCout, aliased as ProgAddress), the Program Memory (prog) storing 64 20-bit instructions, the Decoder (decoder) for producing control signals, the Register File (regs) with 32 8-bit registers for operands and results, the ALU (alu) for arithmetic/logical operations and flag generation, the ROM (rom) with 256 8-bit entries for LDRIND and LDROM operations, and the Debouncer (debounce) to stabilize the handshaking signal into stable_ handshaking signal. Additional components include the Immediate Multiplexer (Alub) selecting the ALU input b from Rdata2, immediate (I[n-3:0]), or rom_dout, and the ROM Address Multiplexer (rom_addr) choosing the ROM address from base_index, Rdata1 + offset, or I[5:0]. The processor interfaces with external inputs such as clk, an active-low reset, and sw(switches), and outputs an 8-bit outport displaying the ALU result. The data path facilitates the flow of addresses, operands, results, ROM data, and switch inputs, incorporating the Alub and rom_addr multiplexers, while the control path generates signals to orchestrate data path operations, managing these multiplexers to ensure proper functionality.The clear dinstinction between the Control and the Data signal can be seen from Figure 1.1.

A snippet of the system Verilog source code for the program memory module is shown is shown below

```
module prog #(parameter Psize = 6, Isize = 20) (
input logic [Psize-1:0] address,
output logic [Isize-1:0] I  // [21:0]
);

logic [Isize-1:0] progMem [0:(1<<Psize)-1];
initial
begin
   for (int i=0; i<2**Psize; i++) begin
      progMem[i] = 20'h000000;  // 24-bit NOP
      end
      progMem[0] = 20'h00000;  // NOP
      progMem[1] = 20'h42800;  // LDSW %5                  0100_00101_00000_000000
      progMem[2] = 20'hA2810;  // BEQ %5, %0, +1           1010_00101_00000_010000
      progMem[3] = 20'h75800;  // MOVSW %11,               0111_01011_00000_000000
        .
        .
end
```

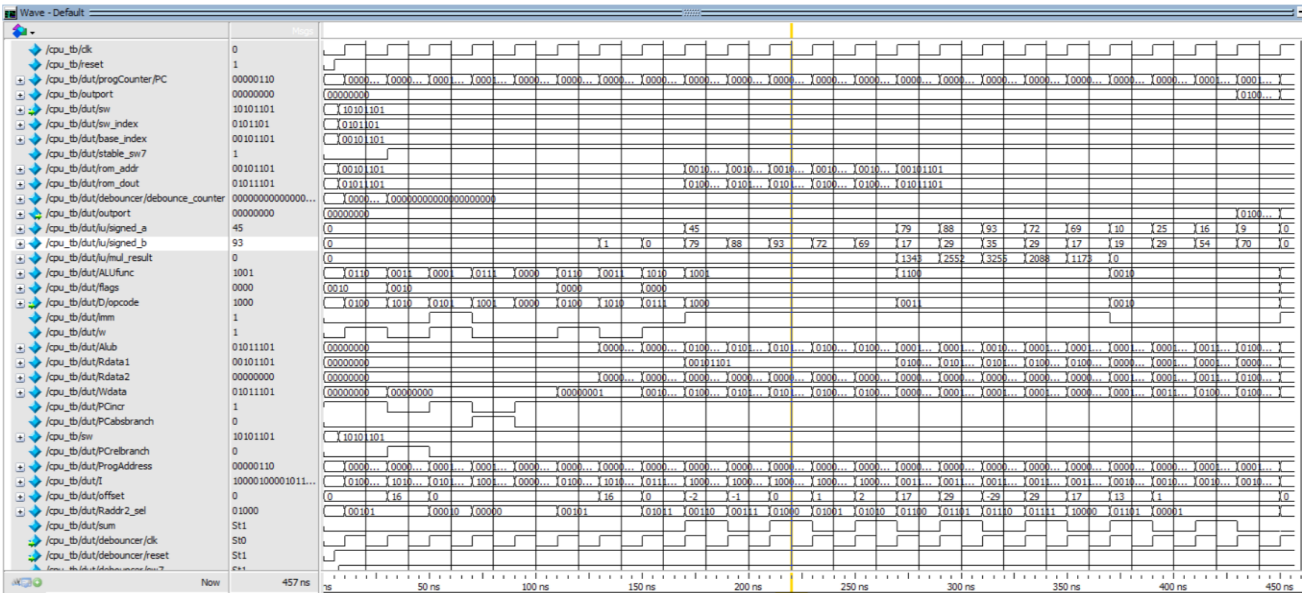

Figure 1.2 The simulation output of the design

The waveform output of the simulation of the modules and the test bench is shown in Figure 2.1 above. After the reset has been deasserted and it is high, the system starts to work and begins at the program counter at zero; since the program counter at 0 is NOP, nothing happens, then the PC goes to 1, and the instruction set there is decoded. As can be seen from the figure above, the opcode has been decoded as "0100"; hence, this is the "LDSW" instruction; since the immediate is set to zero and the write is set to high, the value will be written to a register; the setting of the imm and w is done in the decoder module, which can be seen in the waveform output. The decoder assigns the ALU func as "0110," which is the LSW; hence, in the Arithmetic Logic Unit, the value of the stable handshaking signal is assigned to the result, and since the w is high, it will be written to the register value in the destination register; this means the status of the handshaking signal is stored in the register.

If the handshaking signal is high in the second program, the instruction is decoded and it is decided that it is the BEQ instruction; then, in the decoder, the immediate is zero and the write signal is set to high, the ALUfunc is assigned Rsub. In the ALU, the value in the destination register and the zeroth register is subtracted; if the difference is different from 0, it means the handshaking is asserted; hence, it will not branch; if the difference is zero, it will branch to the 18th program. Then, from the 18th program, it will continue to the 19th and then jump back to the 0th program, and the whole program continues all over again.

Since the handshaking is asserted, it will move to the 3rd program, which loads the index value from the switches and stores it in the destination register; hence, the instruction is decoded in the previous way, and then the ALU func is assigned to movsw. In the Arithmetic Logic Unit, it will assign the values of the switches to the result, and since the write signal is set to high in the decoder, the switch values will be written and hence stored in the destination register.

The program counter now moves to the 4th program; in this program, the value from the ROM is fetched based on the switch value, a.k.a. the index value, which was stored in the 3rd program, and the offset value stored in the offset bits of the instruction. In this program, it is fetching the values two steps back from the base index; to go two steps back, the offset value is assigned all bits zero except the least significant bit, and since it needs to be 254 so that subtraction can be made from the base index value. To be 254, the offset bits need to be 8, but in this case, it is 6; hence, sign bit extension is implemented so that the six-bit opcode will be extended to 8 bits while the two most significant bits are 1. When the value is added, it performs the index value minus two and the index value minus one. The program will continue to program 8, where it fetches different values based on the value of the offsets.

In program 9, it will multiply the value stored in the source register, register 6 in this case, and the kernel value stored in the immediate, and then store the value in the destination register, register 12 in this case. The instruction set is decoded, and the opcode is the MUL operation, and in the decoder, the ALUfunc is assigned MUL_ALU; then, the immediate and write are set to high. Next, in the ALU, the dedicated multiplier is inferred, and the operands, signed_a and signed_b, are received from the Rdata1 and Alub; due to the custom multiplexer switching, the value of Alub is assigned if the opcode is the multiplier one. Then, they are multiplied, and the multiplication will be 16 bits; hence, the bits from the 7th to the 14th are truncated and assigned to the result, and since the write is enabled, the value will be written to the destination register and stored.

The program will continue to program 13 and hence calculate the intermediate multiplications, w[i-2]*k[0], w[i-1]*k[1], w[i]*k[2], w[i+1]*k[3], w[i+2]*k[4], and store them in the destination registers.

In program 14, the instruction is decoded, and the opcode is the ADD instruction; the ALUfunc is assigned the RADD, and the immediate is set to 0, and the write is set to 1. In the ALU, it is going to add the values and store them in the destination register because the write is set to 1; in this program, the value, the intermediate multiplication which is stored in the register 12 from program 9 and the intermediate multiplication which is stored in the register 13 from program 10, are added and stored in register one. The program continues till program 17 by adding the rest of the intermediate values in the register one, and hence implementing the chained addition technique. By using a single register to store the values, it will save the number of registers used, and at program 17, we will find the final result of S[i].

3. FPGA implementation

After the simulation was working perfectly, the .sof file was uploaded to the FPGA and tested, but there was erratic behavior; sometimes it works, sometimes it does not work, and hence, the issues identified are the bouncing effect of the switch, setup and hold time violation, metastability, and unconstrained paths. To solve all these issues, a different approach was taken.

Unlike the push buttons, which have a dedicated Schmitt trigger-based debouncing as the pushbuttons [2], since the slide switches are directly connected to the 5CSEMA5F31C6N System on Chip on the DE1-SoC, the bouncing effect of the switch will be directly sensed. To solve this, there are two solutions: a hardware solution and a software solution. The hardware solution is to add a floating resistor, and the software solution is to use a debouncing logic. For this case, a debouncing logic is implemented. The debouncing logic will count to the set amount of limit; as soon as the counter reaches the limit, the switch will have settled by then, and the value can be referenced from that, which will remove the bouncing effect.

For the timing issue, a system design constraint file needs to be added so that Quartus will have the appropriate constraints, and hence, the setup and hold times are not violated. First, when Quartus was compiled, it showed an output that the timing requirement was not met, so the timing analyzer is opened, and the clock, input, and output constraints are specified; finally, the constraints are written to the SDC file, and the file is added to the timing analyzer so that Quartus references it. After this, it was compiled again, and the timing requirement was fulfilled.

Now, the .sof file is uploaded, and the FPGA is tested. First, the active-low reset signal is set to low, and the index values are set; then, the handshaking signal is set to high; nothing happens, the LEDs are all low because the reset is low. Next, the reset is set to high, the index values are set, then the handshaking signal is set to high; now, the LEDs show some result. It can be confirmed that the system is working overall.

Next, just to make sure everything works, ten index values are selected, which can simulate corner case scenarios, and those are tested; the output on the LEDs was shown to be accurate when compared with the simulation and with precomputed values, so the system on the FPGA is now successfully tested and verified.

4. Conclusion

The embedded processor utilizes an 8-bit data path and a 20-bit instruction size with four types of instruction formats. I have developed techniques and skills in debugging complex systems, enabling me to understand how timing requirements are influenced by high-speed 50 MHz clocks and to address practical challenges such as switch bouncing when working at these high clock speeds. I have gained insight into creating custom instructions for specific applications, such as loading the switch, and comprehend how different modules, including the program counter, arithmetic logic unit, and more, are integrated to form a complete embedded processor. I have learned what it takes to design an embedded processor for a specific application, understanding how my design choices impact the speed, size, power, and consequently the cost of the processor. I am able to apply the skills and knowledge acquired from the high-level synthesis course to implement higher-level optimizations that reduce the complexity and cost of the embedded processor.

To further optimize the overall system, I would aim to reduce the number of registers used, thereby simplifying the instruction set, which will decrease the complexity and cost of the processor as a whole. The speed of the processor can be further enhanced by pipelining the system and resolving hazards through the addition of registers. Additionally, decoding can be simplified by precomputing the decoding within the program module.

1. 1.7. References

Quote the sources of your information. Especially make reference to any sources you used in the development of your code.

[1] Tomasz Kazmierski "Sample SystemVerilog files for picoRISC modules," School of Electronics and Computer Science, University of Southampton, Southampton, U.K., [Online]. Available: https://secure.ecs.soton.ac.uk/notes/elec6234/picomips/

[2] P. Y. K. Cheung, "DE1-SoC User Manual," Dept. of Electrical and Electronic Engineering, Imperial College London, London, U.K., [Online]. Available: http://www.ee.ic.ac.uk/pcheung/teaching/ee2_digital/de1-soc_user_manual.pdf

[3] Intel Corporation, "Inferring Multipliers," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683082/22-1/inferring-multipliers.html