Table of Contents

0. General Information

The zip file we have sent contains this document and some other folders. There are five folders as follows:

1. Verilog Code: This folder contains the Verilog code for the designs.
2. Test benches: This folder contains the test bench that just generates a clock input. The microprocessor is tested using the test bins inside the other folder. With each clock the next instruction is read from the memory and execution takes place accordingly.
3. Test Bins: This folder contains the test cases that you can put within the instruction memory.
4. Synthesis Results: This folder contains the ".syr" files of the microprocessor.
5. Bigger Schematics: This folder contains larger versions of the images used within this document. If the reader wants to get a better look at the designs or the simulation outputs, they can refer to here.

1. Primitives Implemented

In this section I will explain the design of some of the primitives that were used to implement the shifter the program counter and the FPGA implementation of the program counter.

1.1. From the Previous Assignments

From the previous assignments the following primitives were taken directly.

- Binary to Seven Segment Display (In Hex) Encoder
- Seven Segment Display Wrapper (to time multiplex the 4 displays)
- Shift Register (used in debouncer)
- Clock Slower (used for debouncer)
- Debouncer
- Register (to hold the flag values and the instruction)

The following modules are taken from the previous assignments but they are slightly modified.

- Register File
    - Normally the register file was two port read and one port write. But to see the contents of the register 0 on the seven segment displays, I have included a third output port which is not addressable (it always shows the contents of register 0). This port is not used by the microprocessor internally.
- Sign Extender
    - The mode (zero-extend or sign-extend) of the sign extension unit was parameterized, now it is multiplexed by the additional select bit.

- ALU
  - The opcodes for the ALU have been modified to reflect the changes according to the document (lecture9.2cpu_summary) our instructor has provided us.
  - An additional MOV command is added to it.
  - Also I chose to implement the LUI (Load Upper Immediate) operation within the ALU.
- The Shifter Unit
  - In the previous assignment I have implemented a shift unit that could do rotate and arithmetic shifts. For this implementation to reduce the size of the circuit (the previous one is bloated for our purpose), I have downgraded it to handle just standard left and right shift.

1.2. Control Unit

The control unit is a combinational circuit. It has mainly two parts.

1. Jump condition multiplexer:
   a. The 16 different jump conditions are implemented with a multiplexer (case statement).
   b. This value is calculated for all types of instructions. But its value is used only for conditional branch (Bcond) and conditional jump (Jcond) instructions.
   c. The specification directly matches the one within page 7 of the document "lecture9.2cpu_summary".
2. Control Signal Generation:
   a. In this part the control signals for the datapath are generated according the instruction. I mainly have used 9 control signals.
      i. ext_signed, // Sign Extend Unit Mode Select
      ii. bSelect, // ALU B input select (immediate vs register file)
      iii. shftSelect, // Shifter Input Select (immediate vs register file)
      iv. aluSelect, // ALU OpCode Select (instr[15:12] vs instr[7:4])
      v. wregSelect, // Register File Input Select (ALU output vs Shifter Output vs Data Memory Output vs Program Counter Value)
      vi. jmp, // Jump Select
      vii. branch, // Branch Select
      viii. rwren, // Register File Write Enable
      ix. dwren, // Data Memory Write Enable
   b. The status signals that were used by the control unit were:
      i. PSR values (5 of them, mainly ZNFCL flags)
      ii. Instruction

1.3. Memory

The memory was implemented according to the source code given with the assignment. The only difference I have made are the following:

- Read operations are asynchronous. Write operations are synchronous.
- Instruction memory is read only. Data Memory can also be written.

2. Microprocessor Implementation

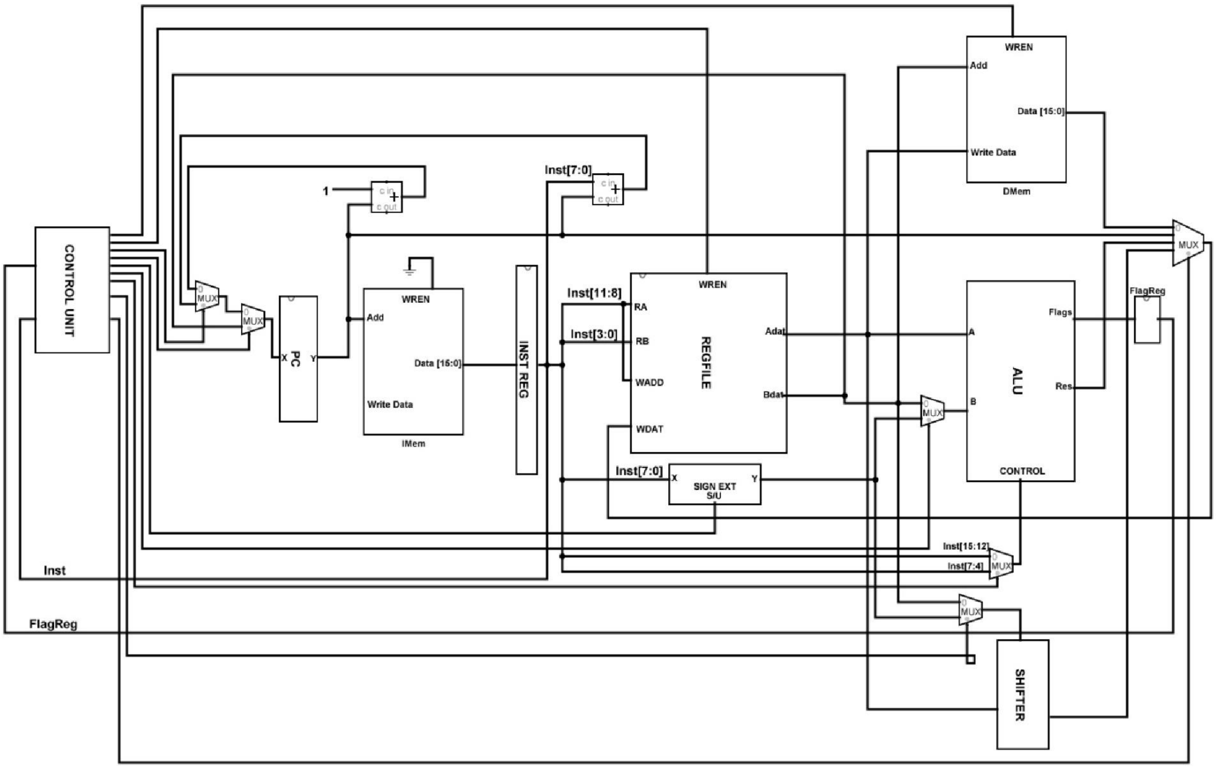
Here we will first explain the design of the simple microprocessor. Then we will show the simulation results.

2.1. Design

The design of the microprocessor is based on the lecture notes given. It is a two clock cycle pipelined design. It consists of the fetch and execute phases. During the fetch phase the program counter will be updated according to the previous homework and the instruction at the memory location pointed by the program counter will be read to the instruction register. The execute phase consists of decoding the instruction, reading the required operands from the register file and executing the operation using the ALU or the shifter unit. The read operation from the register file is asynchronous, so it is done within a single clock cycle. This is required to implement a two clock cycle deep pipeline. If the read operation of the register file was also synchronized then we would need to wait another clock cycle for reading the operands, so our execution would be delayed by another clock cycle. Note that the write operation is still synchronous, since we allow the source and destination registers to be the same. Within the diagram below you will also see that we have an additional register just above the instruction register. This is required since the branch operation is defined as adding to the current value of the program counter. In many other implementations the branch operation is normally added to the already incremented program counter. But since in our design that is not the case, and since we have a pipelined design we need to keep the previous (non-incremented) value of the program counter for this.

Another point with our design is the following. Since our memory write operation takes a clock cycle (it is synchronous) , the memory store operations actually take one additional cycle. But since our instruction set contains only instructions that read or write a single value to/from the memory, this does not cause any problems.

The circuit is as follows. To get a bigger picture the reader may refer to "Bigger Schematics" folder. Also note that the second adder for displacement is written behaviorally. So the schematics shows that it takes the PC and instr[7:0] as input. Actually the instr[7:0] is also sign extended. But I haven't used the sign extension unit under the ALU for this purpose.



2.2. Control Signals

Here I will include a table for the control signals for each operation type. condflag is the output of the multiplexer that uses the PSR flags.

Instruction	Ext_signed	bSelect	shftSelect	aluSelect	wregSelect	Jmp	Branch	rwren	dwren
0000 (Register Ops)	0	1	0	0	2'b10	0	0	1	0
1100	0	0	0	0	2'b00	0	condflag	0	0
0101,1001,1011 (Sign Extended Immediate)	1	0	0	1	2'b10	0	0	1	0
0001, 0010, 0011, 1101, 1111	0	0	0	1	2'b10	0	0	1	0
1000 (LSH)	0	0	0	0	2'b11	0	0	1	0
1000 (LSHI)	0	0	1	0	2'b11	0	0	1	0
0100 (Jcond)	0	0	0	0	2'b00	condflag	0	0	0
0100 (JAL)	0	0	0	0	2'b01	1	0	1	0
0100 (Store)	0	0	0	0	2'b00	0	0	0	1
0100 (Load)	0	0	0	0	2'b00	0	0	1	0

2.3. Simulation

In addition to the test file given to us with the assignment (the one with 45 instructions) I have included additional instructions at the end of it to show the remaining functions of the CPU. ADD, LUI, Jcond and JAL instructions were not tested within those instructions.

The following is the instructions that were added with their lines.

ADD R1 R0	45	0000000001010001	displays hex(19)
LUI hex(AB) R0	46	1111000010101011	displays hex (0)
LSHI hex(-8) R0	47	1000000000011000	displays hex (AB)
MOVI hex(40) R4	48	1101010001000000	
JAL R3 R4	49	0100001110000100	
OR R0 R0	50	0000000000100000	
MOVI hex(01) R0	51	1101000000000001	
MOVI hex(02) R0	52	1101000000000010	
MOVI hex(A0) R0	53	1101000010100000	
JUC R0	54	0100111011000000	
OR R0 R0	55	0000000000100000	
OR R0 R0	56	0000000000100000	
OR R0 R0	57	0000000000100000	
OR R0 R0	58	0000000000100000	
OR R0 R0	59	0000000000100000	
OR R0 R0	60	0000000000100000	
OR R0 R0	61	0000000000100000	
OR R0 R0	62	0000000000100000	
OR R0 R0	63	0000000000100000	
CMPI hex(03) R5	64	1011010100000011	R5 contains 0.

JLT R3	65	0100110011000011	
OR R0 R0	66	0000000000100000	
ADDI hex(01) R5	67	0101010100000001	
MOV R5 R0	68	0000000011010101	(to see R5 content at the output)
JUC R4	69	0100111011000100	
OR R0 R0	70	0000000000100000	

The last part from 64 to 70 actually checks whether 3 is less than the contents of R5. If not R5 is incremented by 1. If it is less than R5 than the execution jumps to 50 (the link from previous JAL operation) and continues from there. On line 65 I have also tested another jump condition that is based on flags (less than). The original test case only contained the equality check (BEQ) which is simpler than the other flags.

Again I have used display to show the contents of the PC and register 0. The simulation was correct and you can find the trace as follows:

```
Simulator is doing circuit initialization process.
Finished circuit initialization process.
20 Program Counter 0, Reg0: 0
40 Program Counter 0, Reg0: 0
60 Program Counter 0, Reg0: 0
80 Program Counter 0, Reg0: 0
95 Reset Is Asserted
100 Program Counter 0, Reg0: 0
120 Program Counter 1, Reg0: 0
140 Program Counter 2, Reg0: 0
160 Program Counter 3, Reg0: 1
180 Program Counter 4, Reg0: 0
200 Program Counter 5, Reg0: 0
220 Program Counter 6, Reg0: 2
240 Program Counter 7, Reg0: 3
260 Program Counter 8, Reg0: 2
280 Program Counter 9, Reg0: 5
300 Program Counter 10, Reg0: 5
320 Program Counter 11, Reg0: 7
340 Program Counter 12, Reg0: 7
360 Program Counter 13, Reg0: 6
380 Program Counter 14, Reg0: 4
400 Program Counter 15, Reg0: 8
420 Program Counter 16, Reg0: 0
440 Program Counter 17, Reg0: 0
```

```
460 Program Counter 18, Reg0: 4
480 Program Counter 19, Reg0: 8
500 Program Counter 20, Reg0: 8
520 Program Counter 21, Reg0: 9
540 Program Counter 22, Reg0: 9
560 Program Counter 23, Reg0: 7
580 Program Counter 24, Reg0: 7
600 Program Counter 25, Reg0: 7
620 Program Counter 26, Reg0: 7
640 Program Counter 27, Reg0: 10
660 Program Counter 28, Reg0: 10
680 Program Counter 29, Reg0: 10
700 Program Counter 40, Reg0: 10
720 Program Counter 41, Reg0: 10
740 Program Counter 42, Reg0: 11
760 Program Counter 43, Reg0: 12
780 Program Counter 44, Reg0: 13
800 Program Counter 45, Reg0: 14
820 Program Counter 46, Reg0: 15
840 Program Counter 47, Reg0: 25
860 Program Counter 48, Reg0: -21760 (AB00 from LUI operation)
880 Program Counter 49, Reg0: 171 (00AB after shifting to the right by 8)
900 Program Counter 50, Reg0: 171
920 Program Counter 64, Reg0: 171
940 Program Counter 65, Reg0: 171
960 Program Counter 66, Reg0: 171
980 Program Counter 67, Reg0: 171
1000 Program Counter 68, Reg0: 171
1020 Program Counter 69, Reg0: 171
1040 Program Counter 70, Reg0: 1
1060 Program Counter 64, Reg0: 1
1080 Program Counter 65, Reg0: 1
1100 Program Counter 66, Reg0: 1
1120 Program Counter 67, Reg0: 1
1140 Program Counter 68, Reg0: 1
1160 Program Counter 69, Reg0: 1
1180 Program Counter 70, Reg0: 2
1200 Program Counter 64, Reg0: 2
1220 Program Counter 65, Reg0: 2
1240 Program Counter 66, Reg0: 2
1260 Program Counter 67, Reg0: 2
1280 Program Counter 68, Reg0: 2
1300 Program Counter 69, Reg0: 2
1320 Program Counter 70, Reg0: 3
1340 Program Counter 64, Reg0: 3
1360 Program Counter 65, Reg0: 3
```

```
1380 Program Counter 66, Reg0: 3
1400 Program Counter 67, Reg0: 3
1420 Program Counter 68, Reg0: 3
1440 Program Counter 69, Reg0: 3
1460 Program Counter 70, Reg0: 4
1480 Program Counter 64, Reg0: 4
1500 Program Counter 65, Reg0: 4
1520 Program Counter 66, Reg0: 4
1540 Program Counter 50, Reg0: 4
1560 Program Counter 51, Reg0: 4
1580 Program Counter 52, Reg0: 4
1600 Program Counter 53, Reg0: 1
1620 Program Counter 54, Reg0: 2
1640 Program Counter 55, Reg0: 160 (Here unconditional jump to hex(A0))
1660 Program Counter 160, Reg0: 160
1680 Program Counter 161, Reg0: 160
1700 Program Counter 162, Reg0: 160
1720 Program Counter 163, Reg0: 160
1740 Program Counter 164, Reg0: 160
1760 Program Counter 165, Reg0: 160
1780 Program Counter 166, Reg0: 160
1800 Program Counter 167, Reg0: 160
1820 Program Counter 168, Reg0: 160
1840 Program Counter 169, Reg0: 160
1860 Program Counter 170, Reg0: 160
1880 Program Counter 171, Reg0: 160
1900 Program Counter 172, Reg0: 160
1920 Program Counter 173, Reg0: 160
1940 Program Counter 174, Reg0: 160
1960 Program Counter 175, Reg0: 160
1980 Program Counter 176, Reg0: 160
2000 Program Counter 177, Reg0: 160
```

As can also be seen from the trace the microprocessor works correctly. The memory, the new register file, the new shifter were also tested extensively, but I have not included them here since they were not required by the assignment.

In addition to this I have also included a test case "instructions2.data" for calculating the Fibonacci number at a specific index. This shows that our design could be used to calculate stuff that can actually be useful.

3. Microprocessor Test Circuit for the FPGA

To test the microprocessor on the FPGA we use the same design as in the previous section. The only things that are added are the debouncer circuit which will be used for the button that will simulate a clock pulse and the seven segment display wrapper that shows the lower 8 bits of the program counter on the left 2 seven segment displays and the lower 8 bits of the register 0 on the right 2 seven segment displays. The design of the debouncer and seven segment wrapper are the same as in the Assignment 3. Interested readers may refer to that document.

4. Synthesis Outputs

	Microprocessor
Advanced HDL Synthesis (n = amount of elements)	• 16x3-bit single-port distributed Read Only RAM (1) • 256x16-bit single-port distributed RAM (2) • 17-bit adder carry in (1) • 17-bit adder (1) • 16-bit up loadable accumulator (1) • Flip-Flops (288) • 16-bit comparator greater (1) • 32-bit comparator greater (4) • 16-bit xor2 (1) • 1-bit 16-to-1 multiplexer (1) • 1-bit 2-to-1 multiplexer (91) • 16-bit 16-to-1 multiplexer (2) • 16-bit 2-to-1 multiplexer (21) • 16-bit 4-to-1 multiplexer (1) • 2-bit 2-to-1 multiplexer (8) • 4-bit 2-to-1 multiplexer (1)

	• 5-bit 2-to-1 multiplexer (1)
Number of Slice Registers	317
Number of Slice LUTs	819
Number used as Logic	691
Number of I/O	34
Clock Period	8.698ns

- Although there are a lot of datapath units, the clock period is small. The pipelined approach helps in this regard.
- In our current design the instruction can be read in a single clock cycle. This may not reflect the reality of actual caches.
- After every Jump or Branch operation we need to insert a NOP (OR R0 R0 for example, or anything that doesn't alter data), else control hazards may occur. I have assumed that this is the case for our design.
- The order of the operands for CMP and CMPI operations on the "lecture9.2cpu_summary" and the test case given in assignment 5 did differ slightly. I choose one specific order. If the order that you are going to test is different than my implementation, I can quickly change it or when using the branch or jump conditions the input can be negated. For example instead of using less than, use greater than or equal etc.

References:

1. BOUN - EE 540 - Spring 2015 - Assignment 4 & 5
2. Computer Organization and Design: the Hardware/Software Interface 5th Edition, David Patterson and John L. Hennessy
3. The Verilog Hardware Description Language, Thomas & Moorby
4. Computer Architecture, Video Lectures by Prof. Anshul Kumar, IIT Delhi
5. www.bing.com