Final Project Report

Judah Ben-Eliezer | Tennyson Cheng

Introduction:

The multimedia unit design we made was designed with simplicity and speed in mind. The processor is designed in such a way that the maximum number of operations are synchronized, and it takes advantage of pipelining to ensure that no component is idle at any time.

ALU:

For the ALU, we used a behavioral model with a case? statement to encapsulate the opcodes. The additional file ALU_functions.vhd contains operations that are used repeatedly in the ALU, so as to avoid repetition. The ALU is timeless, with output sensitive to any change in the input vectors.

Register File:

For the register file, the biggest challenge was getting the simulator to accept writing from different source signals without driving the file to the std_logic value 'X'. In the end, we used a behavioral model with an infinite while loop rather than a process sensitive to changing inputs. Instead, the while loop merely halts while waiting for change. This way we were able to use a variable rather than a signal to model the register file. Like the ALU, it is timeless.

Instruction Buffer:

This was nothing complicated, just a simple buffer that writes the input program upon reset and writes the instruction pointed to by the program counter. Again, this module operates without a notion of time.

Forwarding Unit:

For the forwarding unit, we wanted to minimize the times when it would be used as it is likely to be time consuming. Thus for operations that don't use the register inputs to the ALU, the comparison is avoided completely. For those that do, the forwarding unit compares the addresses of the input vectors to that of the ALU output destination. If there is a match, the data is forwarded so the ALU can use the most recent value of the register.

MMU:

This module contains the full pipeline. We implemented it with a structural design, using the previously mentioned components as well as three registers, REG0, REG1, and REG2, to save the intermediate values in the pipeline. On each clock cycle, an instruction is read into REG0, REG0's value is put into the register file and the appropriate registers are read into REG1, REG1's values are put though the forwarding unit into the ALU and the result is written into REG2, and finally REG2's value is written back into the register file.

Testbench:

The testbench is a simple module that initializes a MMU module, and passes in an input from a text file containing opcodes to test on the MMU. It also reads the status of the pipeline on each clock cycle, and writes the output to a file results.txt.

Assembler:

We chose to write the assembler in python because it was simple. The format for writing programs is as follows:
Li instructions: li immediate, offset(register)
R4 instructions: op rd, r1, r2, r3
R3 instructions: op rd, r1, r2
It is fairly strict with syntax, each register name must be preceded by the letter r and the opcode must be valid or the compiler will simply fail with a format error.

Results File:

The results are written by the testbench into a text file. At each clock cycle, the testbench writes the following signals: PC, instr0, instr1, instr2, write_enable, li, write_addr, and write_data, as well as the registers in the pipeline rf_out, fu_in, and alu_in.

Conclusions:

The pipeline is definitely a great way to speed up a processor to handle operations in a shorter time period. While it requires a lot of extra circuitry to avoid hazards and save registers, it can process 4x the number of instructions in the same time. Some changes to improve it would be to make opcodes for easier loading, which would be especially helpful for testing purposes. Either that, or make the instruction buffer bigger. The test program shown in the submission contains mostly li instructions to make the register file more interesting. In any case, we were able to effectively leverage pipelining to make a processor capable of handling lots of data at high speed.
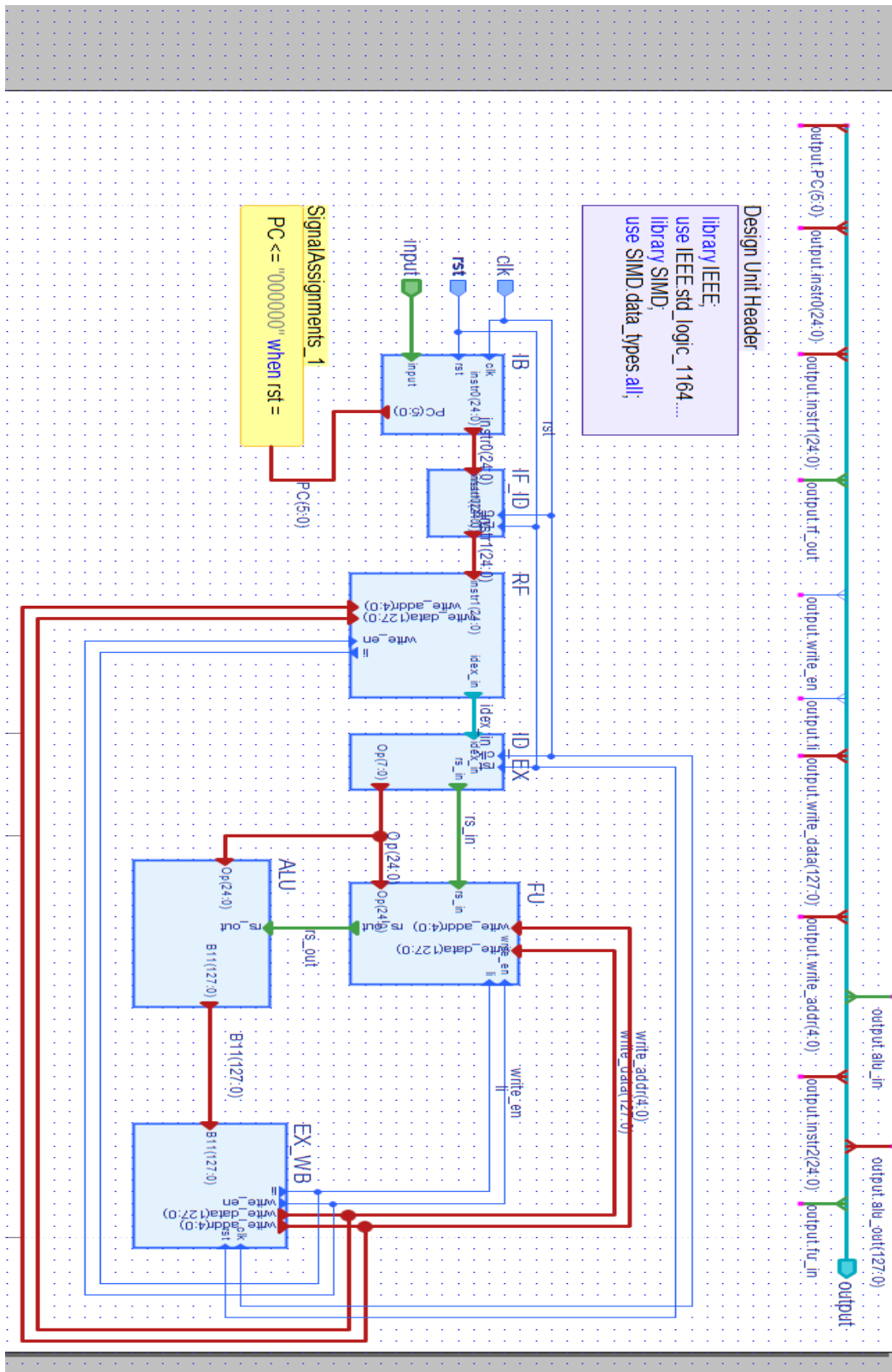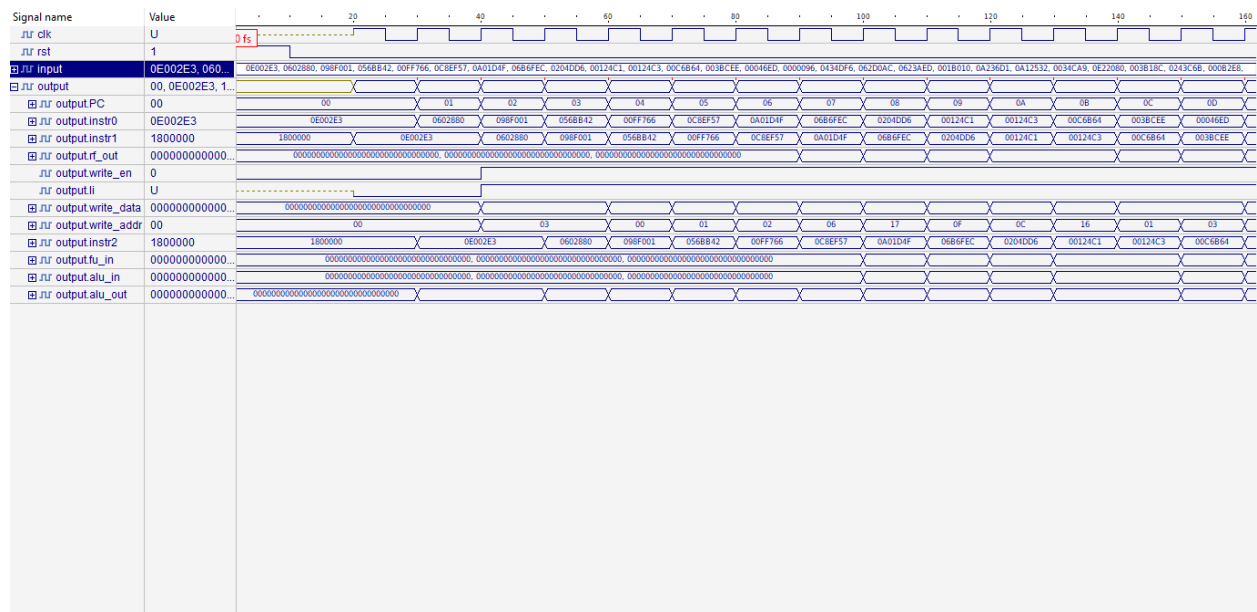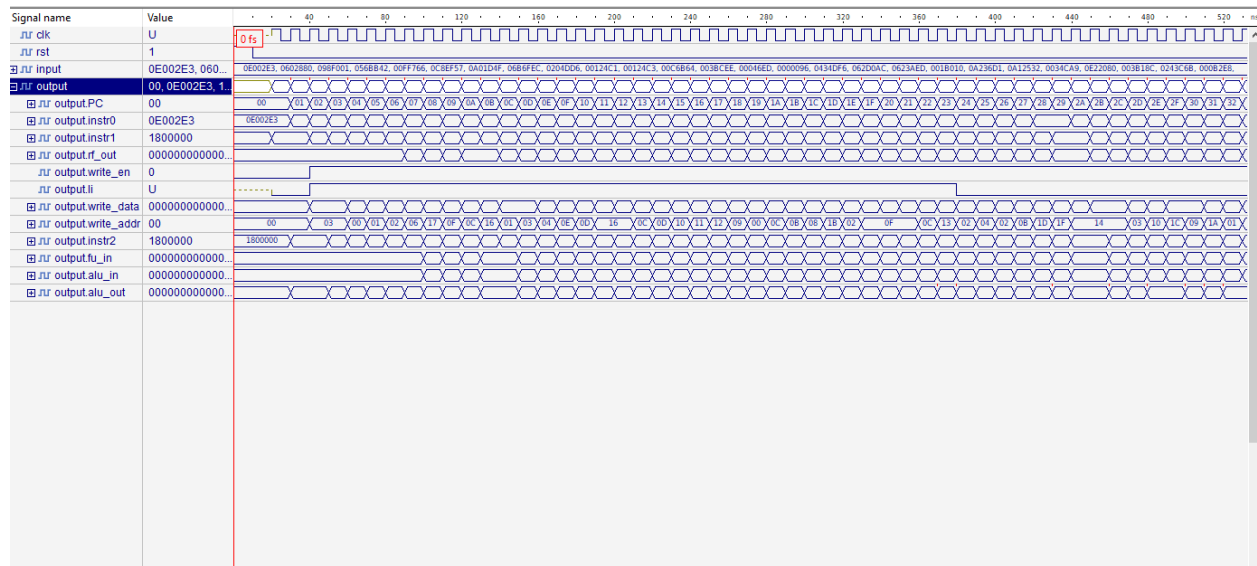
Figure 1: Block diagram

Figure 2: Waveform High Level View

Figure 3: Waveform Detail

```
li 23, 7(r3)
li 324, 3(r0)
li 837492, 4(r1)
li 832974, 2(r2)
li 98234, 0(r6)
li 18298, 6(r23)
li 234, 5(r15)
li 23423, 3(r12)
li 622, 1(r22)
li 2342, 0(r1)
li 2342, 0(r3)
li 25435, 0(r4)
li 7655, 0(r14)
li 567, 0(r13)
li 4, 0(r22)
li 6767, 2(r22)
li 5765, 3(r12)
li 4567, 3(r13)
li 3456, 0(r16)
li 4534, 5(r17)
li 2345, 5(r18)
li 6757, 0(r9)
li 4356, 7(r0)
li 7564, 0(r12)
li 8675, 1(r11)
li 1431, 0(r8)
li 1234, 1(r27)
li 453, 0(r2)
li 2324, 1(r15)
li 834, 4(r15)
li 9283, 7(r15)
li 2345, 3(r12)
li 9823, 2(r19)
simals r2, r4, r8, r3
simahs r4, r14, r3, r7
simsls r2, r1, r0, r5
simshs r11, r23, r16, r18
slimals r29, r28, r17, r13
slimahs r31, r19, r22, r19
slimsls r20, r30, r10, r10
slimshs r20, r11, r0, r1
nop
ah r3, r4, r5
```

```
ahs r16, r15, r12
bcw r28, r21, r6
cgh r9, r24, r25
clz r26, r27, r2
max r1, r2, r3
min r2, r9, r22
msgn r14, r22, r18
popcnth r3, r5, r9
rot r14, r9, r0
rotw r18, r19, r11
shlhi r0, r28, r22
sfh r1, r2, r7
sfhs r0, r0, r0
xor r19, r31, r30
nop
nop
nop
nop
nop
nop
nop
```

```
li 5765, 7(r3)
li 3204, 3(r20)
li 576, 4(r1)
li 5607, 3(r22)
li 98234, 0(r6)
li 18298, 6(r23)
li 234, 5(r15)
li 293, 3(r12)
li 622, 1(r22)
li 5675, 3(r1)
li 2342, 0(r3)
li 4234, 10(r4)
li 7655, 0(r4)
li 567, 0(r13)
li 4, 0(r22)
li 25323, 2(r12)
li 5765, 3(r12)
li 25553, 3(r13)
li 3456, 0(r26)
li 25653, 5(r17)
li 2345, 4(r18)
li 234, 0(r25)
li 4234, 7(r0)
li 346234, 0(r12)
li 234234, 1(r11)
li 1431, 0(r3)
li 2334, 1(r27)
li 243234, 0(r2)
li 24634, 6(r1)
li 24323, 4(r5)
li 9283, 7(r15)
li 243342, 3(r2)
li 9823, 2(r19)
simals r2, r4, r2, r3
simahs r4, r14, r3, r27
simsls r23, r1, r0, r5
simshs r11, r23, r16, r8
slimals r29, r28, r17, r13
slimahs r1, r1, r2, r19
slimsls r22, r30, r10, r10
slimshs r0, r11, r0, r1
nop
ah r13, r14, r5
```

```
ahs r16, r15, r12
bcw r28, r2, r6
cgh r19, r4, r25
clz r26, r7, r12
max r11, r2, r3
min r21, r9, r22
msgn r14, r2, r18
popcnth r3, r5, r9
rot r14, r9, r0
rotw r18, r19, r11
shlhi r0, r18, r22
sfh r1, r20, r7
sfhs r0, r0, r0
xor r19, r11, r30
nop
nop
nop
nop
nop
nop
nop
```

Stimulus.txt

0111000101101000010100011
0011000011001000010010100
0100000000100100000000001
0011000101011110011110110
0000011111111011101100110
0110010001110111101010111
0101000000001110101001111
0011000000010010010101100
0001000000100110111010110
0011000101100010101100001
0000000010010010011000011
0010000100001000101000100
0000000111011110011100100
0000000000100011011101101
0000000000000000010010110
0010011000101110101101100
0011000101101000010101100
0011011000111101000101101
0000000011011000000011010
0101011001000011010110001
0100000010010010100110010
0000000000011101010110010
0111000100001000101000000
0000010010000111111101100
0001100100101111110101011
0000000001011001011100011
0001000010010001111011011
0000101101100010010100010
0110011000000011101000001
0100010111110000001100101
0111001001000100001101111
0011101101101001000100010
0010001001100101111110011
1000000011000100010000010
1000111011000110111000100
1001000101000000000110111
1001101000100001011101011
1010001101100011110011101
1010110011000100000100001
1011001010010101111010110
1011100010000001011000000
1100000000000000000000000
1100000001001010111001101

```
11000000010011000111110000
11000000110011000010111100
11000001001100100100100011
11000001010110000111111010
11000001100001100010010011
11000001111011001001110101
11000010001001000010011110
11000010010100100101000011
11000010100000001001011101
11000010110101110011100010
11000011001011010010000000
11000011010011110100000001
11000011100000000000000000
11000011111111001011100111
11000000000000000000000000
11000000000000000000000000
11000000000000000000000000
11000000000000000000000000
11000000000000000000000000
11000000000000000000000000
11000000000000000000000000
```

Results: see file