*Course No. ELEC 5200/6200*

*Computer Architecture and Design*

Project: Part-5

Date: April 21, 2017

**

*Prepared by*

*Md. Mahabubul Alam*

*Banner ID: 903937005*

*Graduate Student, Electrical Engineering*

*Auburn University*

*Benjamin Cyr*

*Banner ID: 903466318*

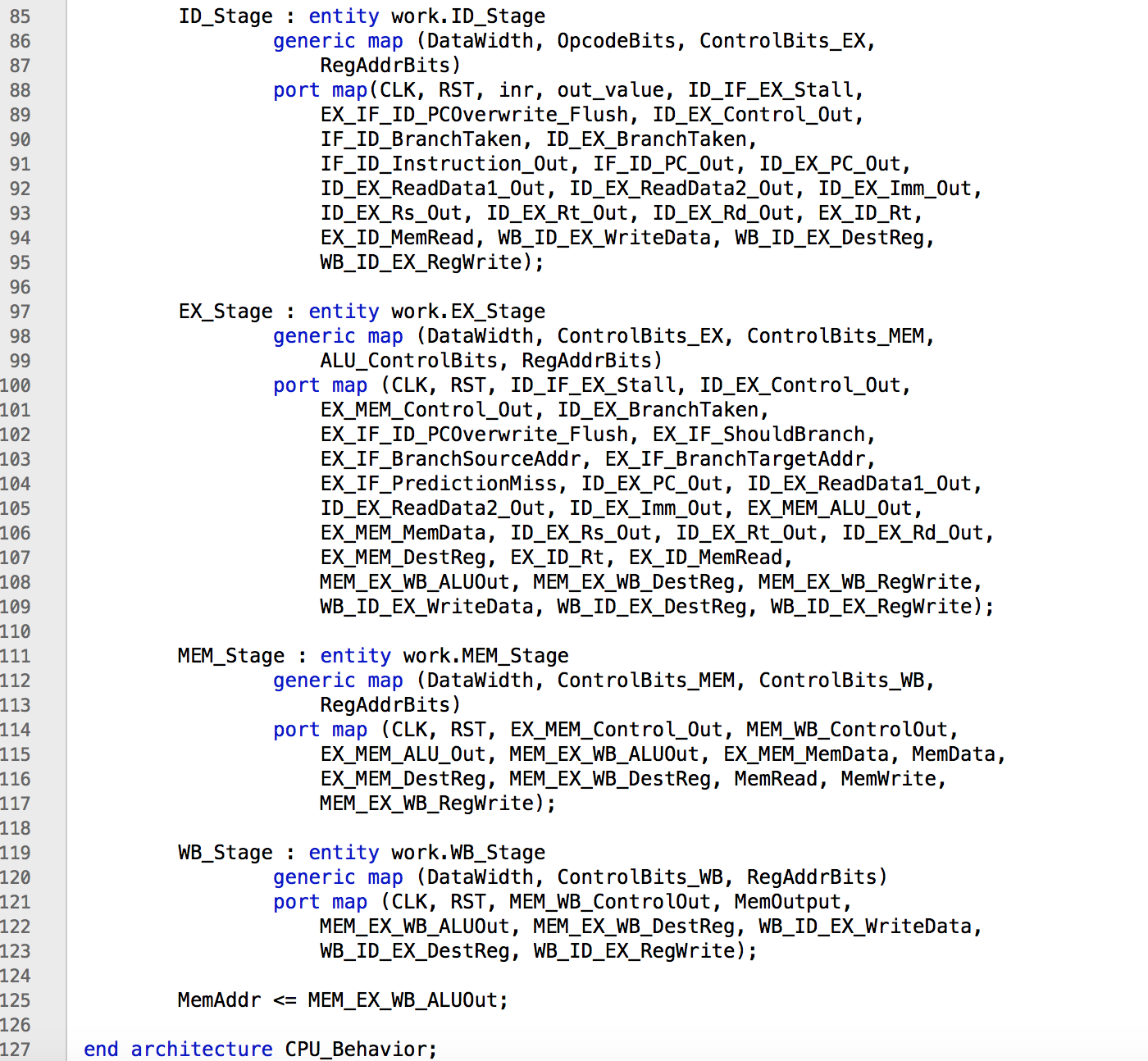
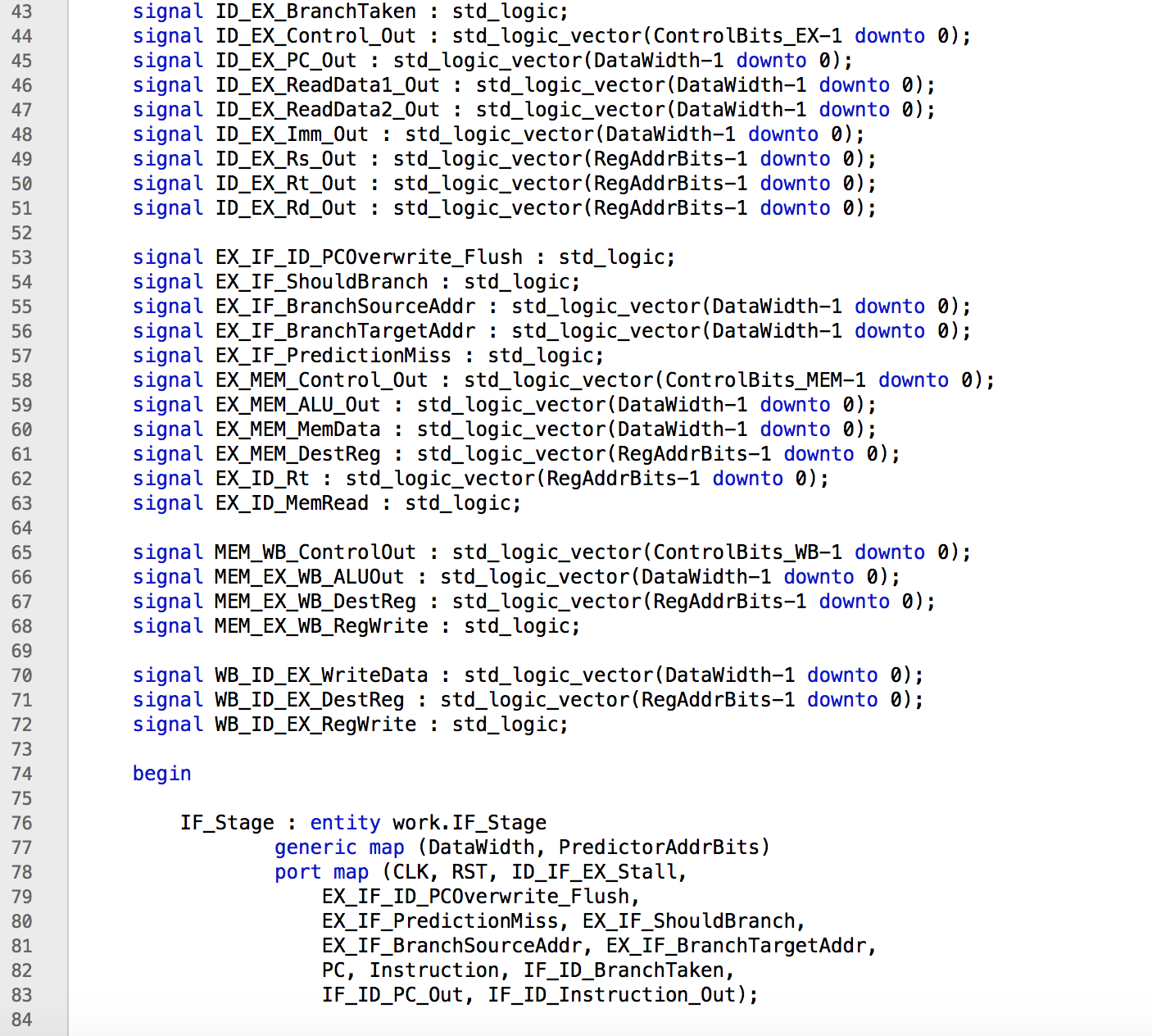
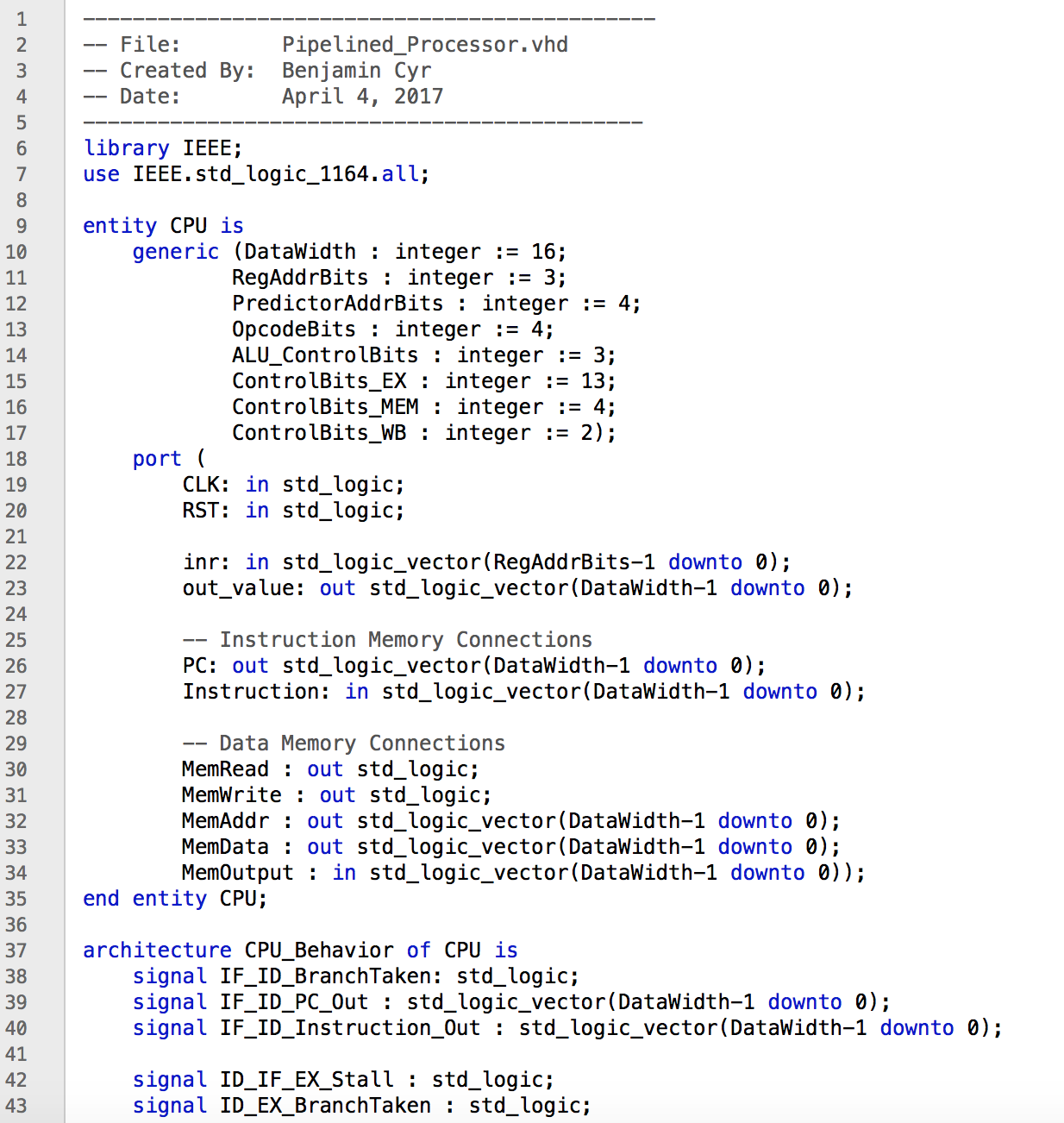
*Senior, Computer Engineering*

*Auburn University*

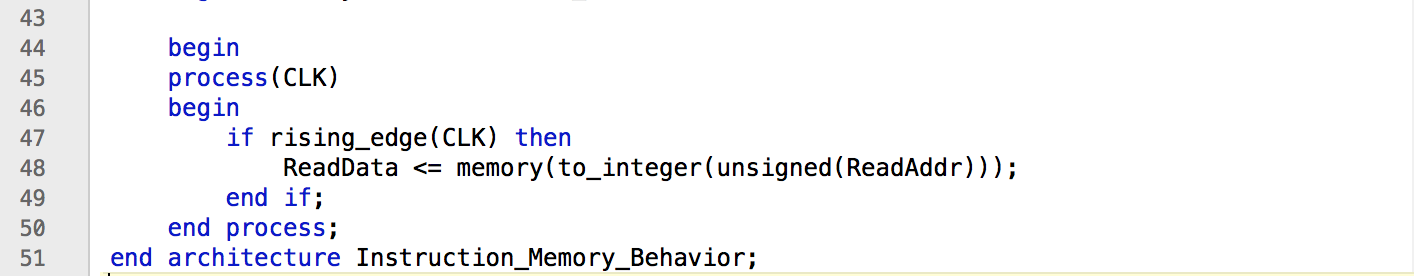
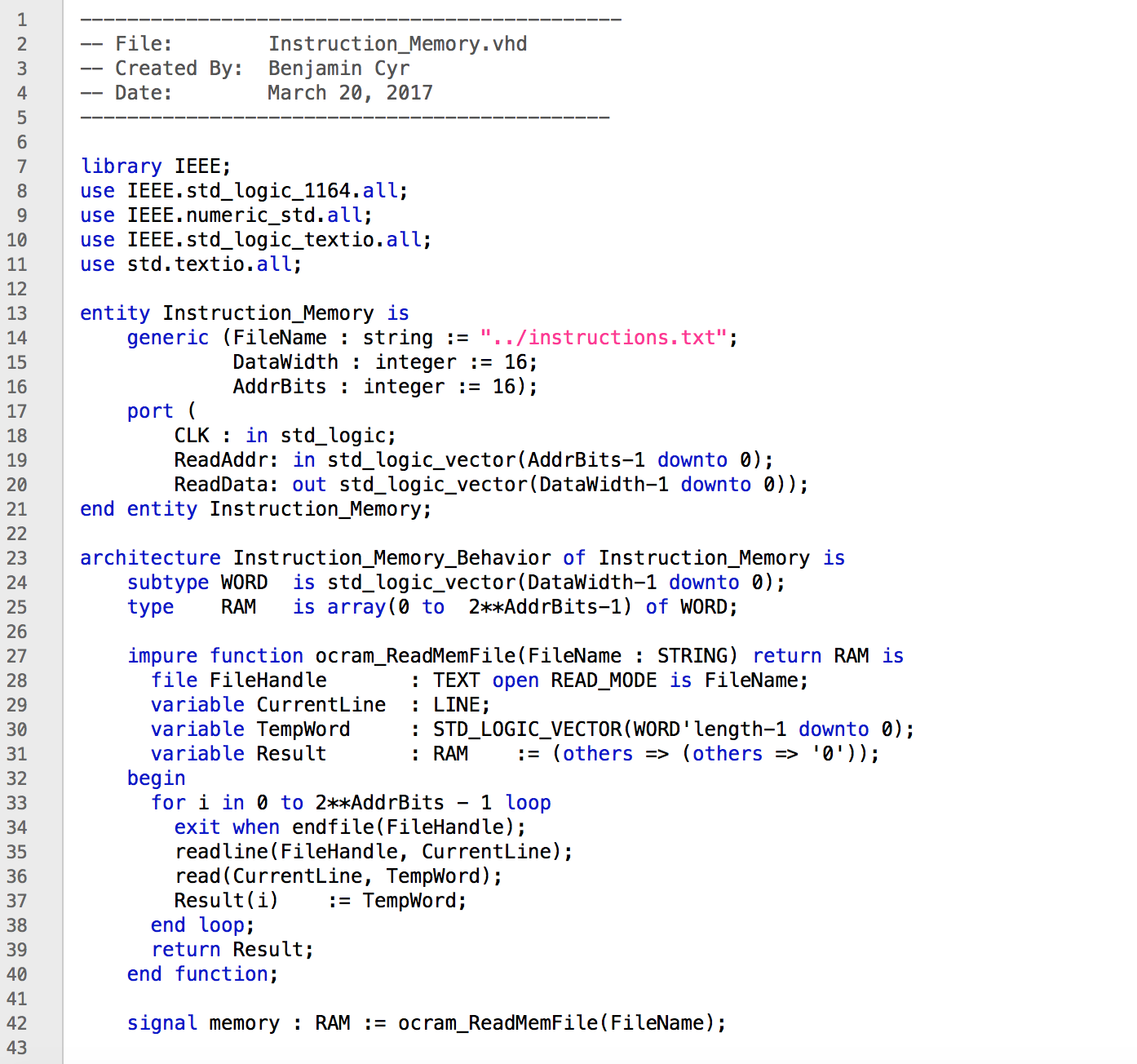
***Source Code:***

This part of the project contained a few new VHDL models to connect the CPU with the data and instruction memories. These models are shown below.

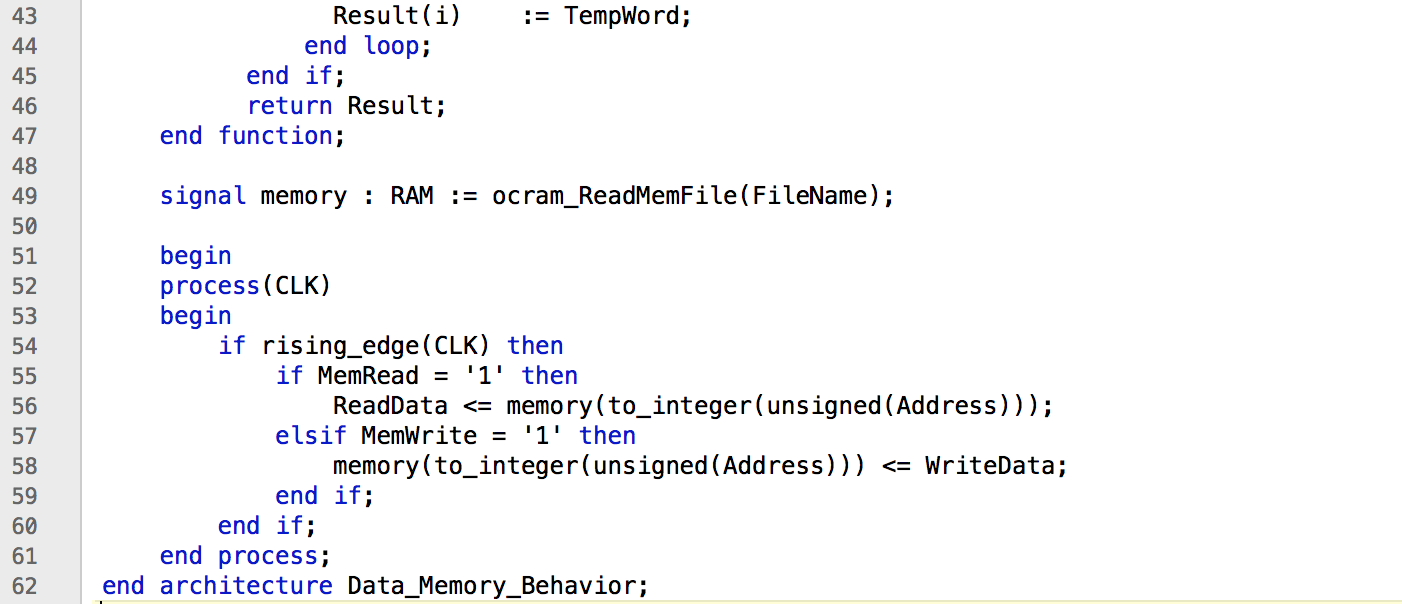
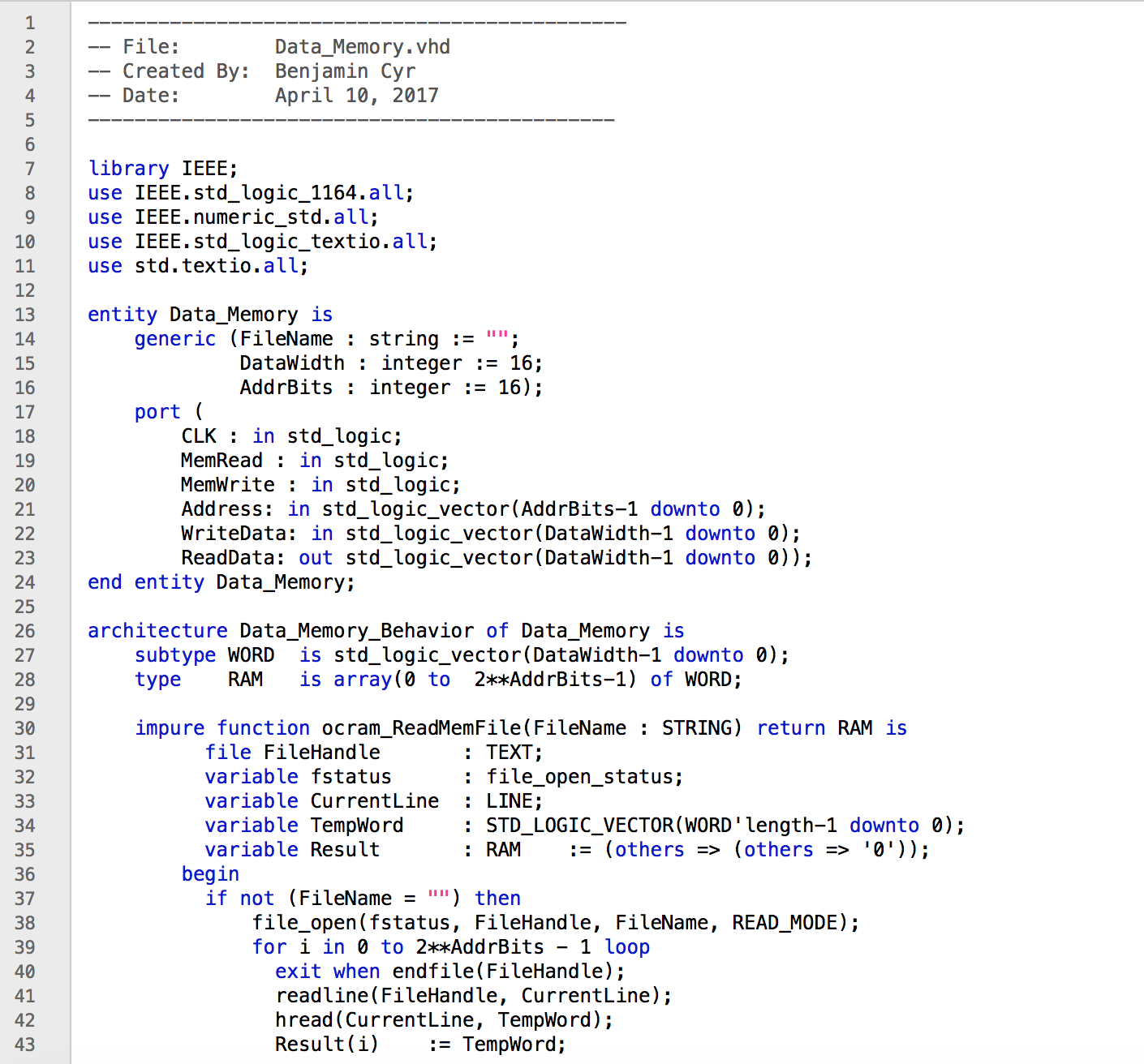
CPU.vhd: Modified from the VHDL File from Part 4. Contains all the stages of the CPU but does not contain any memory. Instruction and Data Memory must be added in a higher level.



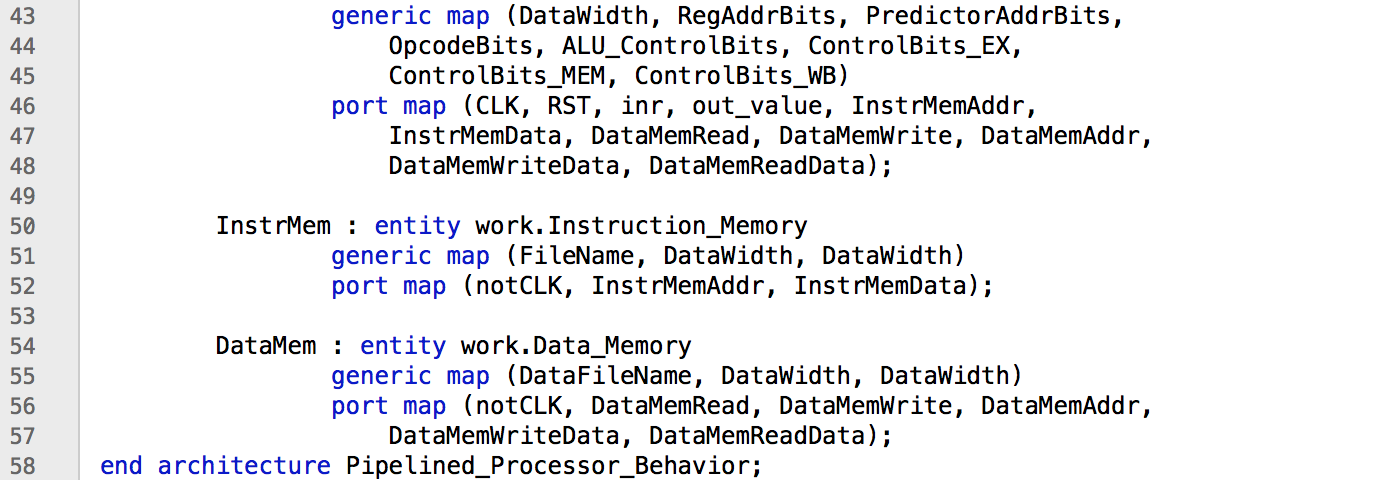
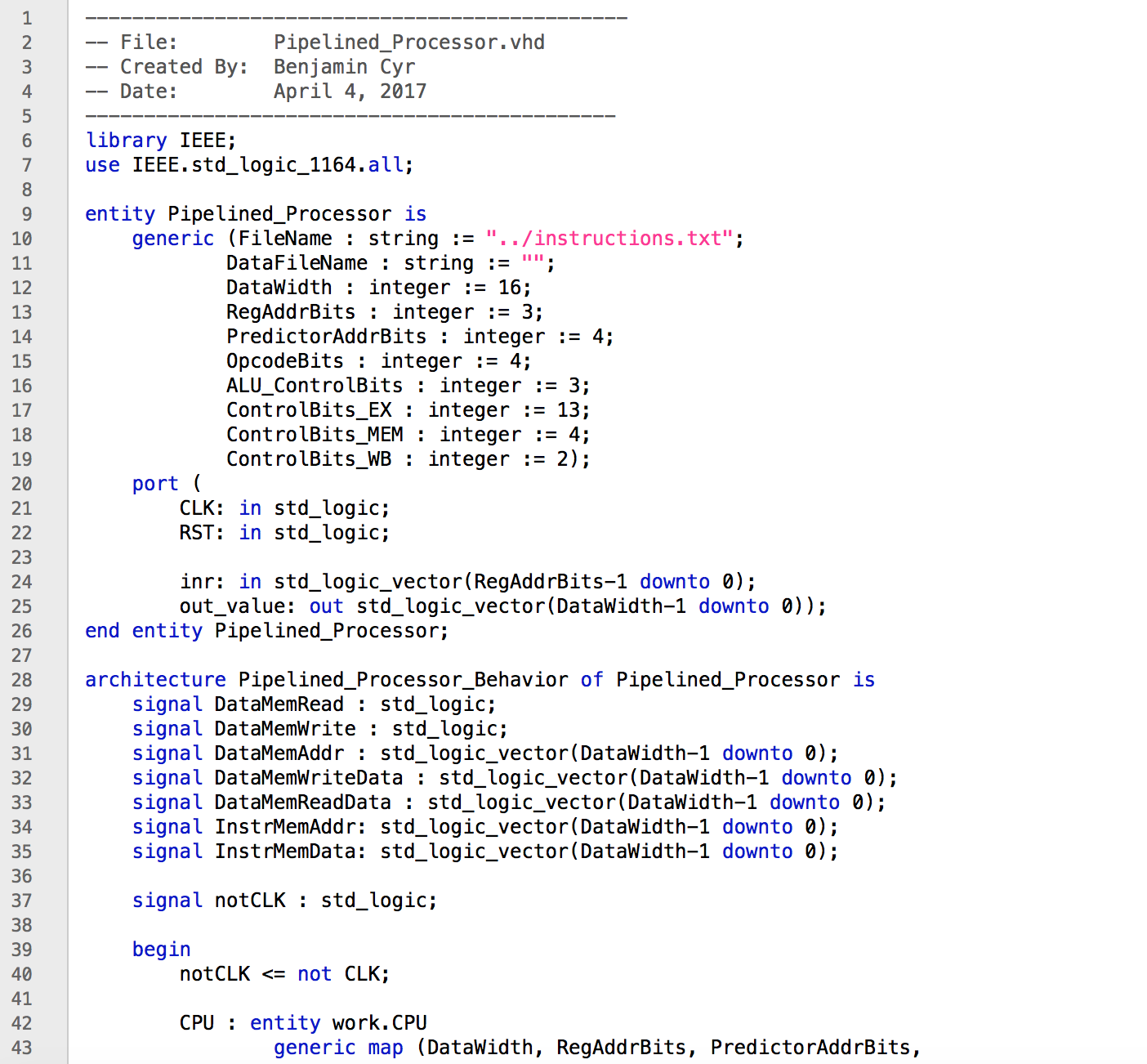
Instruction\_Memory.vhd: The Instruction Memory VHDL file that is outside of the CPU. It is instantiated in block RAM without using the tools for the Altera board. It is programmed to an initial state by reading a text file. The filename is set as a generic.



Data\_Memory.vhd: The Data Memory VHDL file that is outside of the CPU. It is instantiated in block RAM without using the tools for the Altera board. It can be initialized by giving a filename as a generic. If left empty, it programs the memory to all zeros.

******

Pipelined\_Processor.vhd: The top-level model of the processor. Includes the CPU, data memory, and instruction memory modules.



***Simulation Results:***

Out of all the instructions, the only two instructions that have not been tested are Load Word and Store Word. Test benches were written for both instructions and were simulated to show that the Pipelined Processor functioned correctly. The results of the simulations are shown below. The left side shows the instructions that were loaded into instruction memory, while the right side shows the values of the registers after execution.

***LW Simulation Results:***

|  |  |  |
| --- | --- | --- |
| Time | inr | out\_value |
| 110 ns | 0 | 0000 |
| 120 ns | 1 | 0000 |
| 130 ns | 2 | aaaa |
| 140 ns | 3 | 0002 |
| 150 ns | 4 | 1234 |
| 160 ns | 5 | 0002 |
| 170 ns | 6 | 0000 |
| 180 ns | 7 | 0000 |

Initial RAM:

0x0000: AAAA

0x0001: 0002

0x0002: 1234

LW $2, 0($0)

LW $3, 1($0)

LW $4, 0($3)

LW $5, -1($3)

HALT

***SW Simulation Results:***

|  |  |  |
| --- | --- | --- |
| Time | inr | out\_value |
| 110 ns | 0 | 0000 |
| 120 ns | 1 | ffff |
| 130 ns | 2 | 0009 |
| 140 ns | 3 | ffff |
| 150 ns | 4 | 000a |
| 160 ns | 5 | 0000 |
| 170 ns | 6 | 0000 |
| 180 ns | 7 | 0000 |

ADDI $1, $0, -1

ADDI $2, $0, 10

SW $1, 0($2)

SW $2, -1($2)

LW $3, 0($2)

ADD $2, $2, $3

LW $4, 0($2)

HALT

***Test Program:***

Finally, a test program for the demo was written and compiled. The program was designed to be something practical to show that the processor can be useful. A selection sort algorithm was chosen, as sorting is one of the most well-known problems in computing. The C equivalent of the program is shown below:

C Equivalent:

int main() {

int size;

int \*array;

itr\_arr(); // Shows the contents of the first 6 memory locations for demo

size = \*(0x0000);

array = 0x0001;

sort(size, array);

itr\_arr(); // Shows the contents of the first 6 memory locations for demo

for(;;);

}

void sort(int size, int\* array){

int i, j, high, temp;

for (i = 0; i < size - 2; i++) {

high = i;

for (j = i+1; j < size - 1; j++) {

if (array[j] > array[high]) {

high = j;

}

}

temp = array[high]

array[high] = array[i]

array[i] = temp;

}

}

The program functions by reading the first value in RAM. This is the size of the array to sort. The next locations in RAM are the values that the program should sort. It then runs an in-place selection sort algorithm, replacing the contents of RAM with the sorted list. The assembly and binary of the program is shown below:

Assembly Function:

MAIN: JL ITR\_ARR ; Iterate through unsorted array

LW $2, 0($0) ; Load Size of Array

ADDI $3, $0, 1 ; Pointer to start of array

JL SORT ; Sort array

JL ITR\_ARR ; Iterate through sorted array

HALT

ITR\_ARR: LW $1, 1($0) ; Show RAM contents

LW $2, 2($0)

LW $3, 3($0)

LW $4, 4($0)

LW $5, 5($0)

LW $6, 6($0)

JR $7

SORT: SW $7, 31($0) ; Push LR to "stack"

ADD $4, $0, 0 ; $4 = 0

LOOP1: ADD $1, $4, $0 ; $1 = $4

ADDI $5, $4, 1 ; $5 = $4 + 1

SW $4, 30($0) ; Push i to "stack"

LOOP2: ADD $6, $5, $3 ; $6 = &array[$5]

LW $6, 0($6) ; $6 = array[$5]

ADD $7, $1, $3 ; $7 = &array[$1]

LW $7, 0($7) ; $7 = array[$1]

SLT $4, $7, $6 ; if (array[$5] > array[$1]){

BEQ $4, $0, END2

ADD $1, $5, $0 ; $1 = $5}

END2 : ADDI $5, $5, 1 ; $5++

BNE $5, $2, LOOP2 ; for($5 < SIZE)

LW $4, 30($0) ; Pop i from "stack"

ADD $5, $1, $3 ; $5 = &array[$1]

ADD $6, $4, $3 ; $6 = &array[$4]

LW $1, 0($5) ; $1 = array[$1]

LW $7, 0($6) ; $7 = array[$4]

SW $1, 0($6) ; array[$4] = array[$1]

SW $7, 0($5) ; array[$1] = array[$4]

ADDI $4, $4, 1 ; $4++

ADDI $6, $2, -1 ; $6 = SIZE - 1

BNE $4, $6, LOOP1 ; for($4 < SIZE - 1)

LW $7, 31($0) ; Pop LR from "stack"

JR $7

The assembly had to be assembled by hand into binary, and the binary file that was loaded into instruction memory is shown below:

Binary:

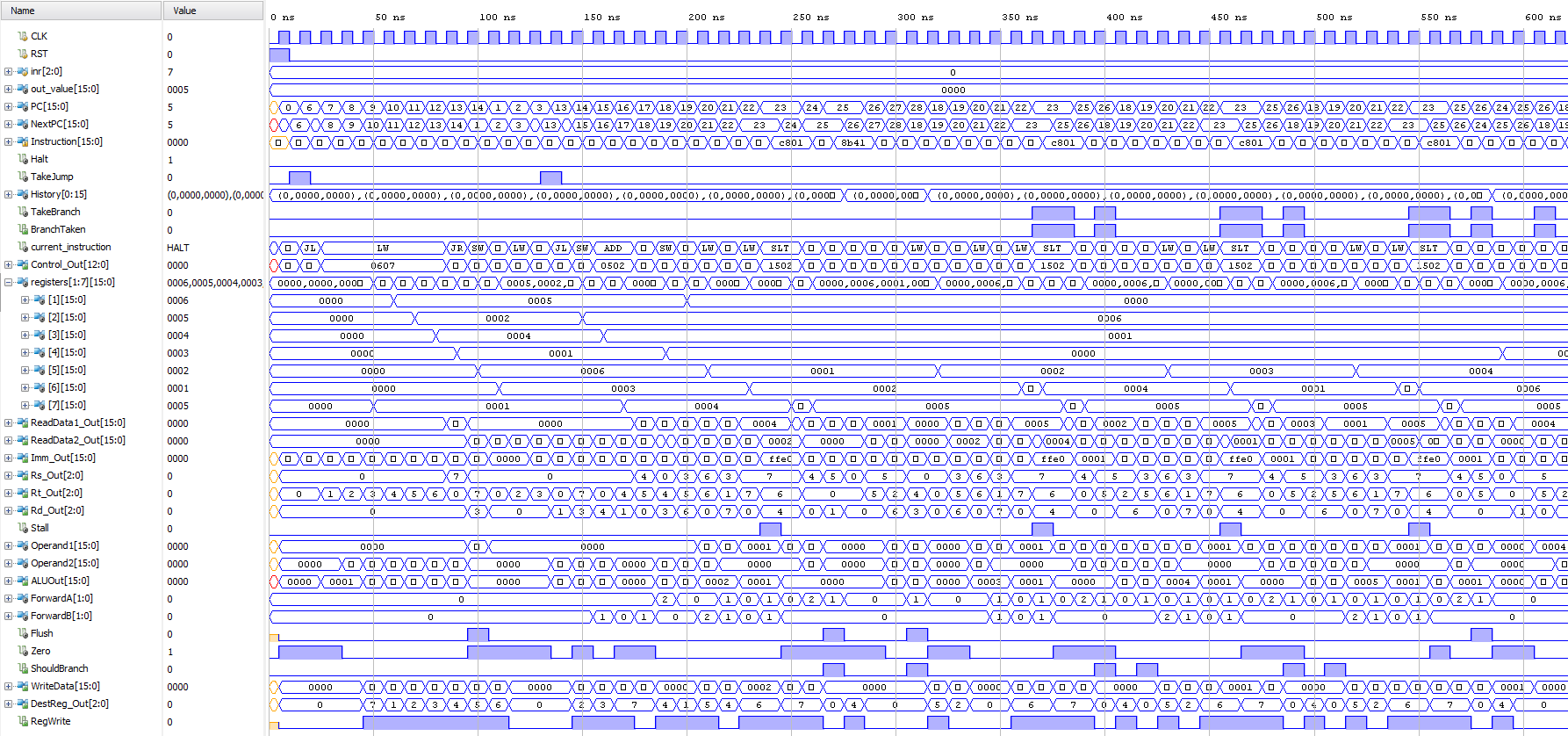
|  |  |  |
| --- | --- | --- |
| 0 | JL 6 | 1101000000000110 |
| 1 | LW $2, 0($0) | 1001000010000000 |
| 2 | ADDI $3, $0, 1 | 1000000011000001 |
| 3 | JL 13 | 1101000000001101 |
| 4 | JL 6 | 1101000000000110 |
| 5 | HALT | 0000000000000000 |
| 6 | LW $1, 1($0) | 1001000001000001 |
| 7 | LW $2, 2($0) | 1001000010000010 |
| 8 | LW $3, 3($0) | 1001000011000011 |
| 9 | LW $4, 4($0) | 1001000100000100 |
| 10 | LW $5, 5($0) | 1001000101000101 |
| 11 | LW $6, 6($0) | 1001000110000110 |
| 12 | JR $7 | 1111111000000000 |
| 13 | SW $7, 31($0) | 1010000111011111 |
| 14 | ADD $4, $0, 0 | 0001000000100000 |
| 15 | ADD $1, $4, $0 | 0001000100001000 |
| 16 | ADDI $5, $4, 1 | 1000100101000001 |
| 17 | SW $4, 30($0) | 1010000100011110 |
| 18 | ADD $6, $5, $3 | 0001011101110000 |
| 19 | LW $6, 0($6) | 1001110110000000 |
| 20 | ADD $7, $1, $3 | 0001011001111000 |
| 21 | LW $7, 0($7) | 1001111111000000 |
| 22 | SLT $4, $7, $6 | 0101111110100000 |
| 23 | BEQ $4, $0, 1 | 1100100000000001 |
| 24 | ADD $1, $5, $0 | 0001101000001000 |
| 25 | ADDI $5, $5, 1 | 1000101101000001 |
| 26 | BNE $5, $2, -9 | 1011101010110111 |
| 27 | LW $4, 30($0) | 1001000100011110 |
| 28 | ADD $5, $1, $3 | 0001001011101000 |
| 29 | ADD $6, $4, $3 | 0001100011110000 |
| 30 | LW $1, 0($5) | 1001101001000000 |
| 31 | LW $7, 0($6) | 1001110111000000 |
| 32 | SW $1, 0($6) | 1010110001000000 |
| 33 | SW $7, 0($5) | 1010101111000000 |
| 34 | ADDI $4, $4, 1 | 1000100100000001 |
| 35 | ADDI $6, $2, -1 | 1000010110111111 |
| 36 | BNE $4, $6, -22 | 1011100110101010 |
| 37 | LW $7, 31($0) | 1001000111011111 |
| 38 | JR $7 | 1111111000000000 |

Finally, the program was simulated with some initial values in data memory. The initial values used in the simulation are shown below, and some of the results of the simulation are shown on the next page:

Initial RAM values:

|  |  |
| --- | --- |
| Address | Data |
| 0 | 6 |
| 1 | 5 |
| 2 | 2 |
| 3 | 4 |
| 4 | 1 |
| 5 | 6 |
| 6 | 3 |

The first part of the simulation is shown in the screenshot below. The important information to see is that the initial contents of RAM is read into registers $1-$6 as “5, 2, 4, 1, 6, 3”. Then the sorting algorithm begins.



The last part of the simulation is shown in the screenshot below. The important information to see is that the final values in registers $1-$6 now show “6, 5, 4, 3, 2, 1”. These are the same addresses in memory, but they have been sorted. The program halted correctly, and took around 275 cycles to sort 6 numbers.

