IEEE Symposium on FPGAs for Custom Computing Machines, Apr.17-19,1996, Napa,California (FCCM'96)

Aizup – A Pipelined Processor Design and Implementation on XILINX FPGA Chip

Yamin Li and Wanming Chu

Computer Architecture Laboratory
The University of Aizu
Aizu-Wakamatsu, 965-80 Japan

`yamin@u-aizu.ac.jp, w-chu@u-aizu.ac.jp`

Abstract

This paper describes a pipelined processor (named Aizup) design and implementation for the exercise of Computer Architecture/Organization Education at the University of Aizu. The Aizup pipeline has four stages and deals with data dependency and control dependency. The Aizup was designed at Cadence environment and implemented on Xilinx XC4006PC84 FPGA chip. We ask students to design the processor, to perform functional simulations, to implement the design on the chip, and to measure the chip with Logic Analyzer. The exercise course is helpful to students to understand the operations of pipelined processors and to master the design methodologies and the use of measuring instruments.

1 Introduction

In the University of Aizu, there are more than sixty sets of exercise equipments for Computer Architecture/Organization Education [2] [3]. One set consists of a SUN workstation with installed Cadence/Xilinx design tools [4], an evaluation board with mounted Xilinx XC4006PC84 FPGA chip [5], and a Logic Analyzer.

We developed a RISC pipelined model (Aizup) and implemented it on the Xilinx chip for Computer Architecture/Organization Education [9]. The contents of the exercise include analysis of processor instruction set, design of datapath and control unit, circuit schematic, functional simulation, netlist generation, pin assignment, placement and routing, Intel mcs-format file generation, file download, and measurement and check of the chip [6]. Because students have mastered the use of Cadence in Logic Circuit Design course, they can finish the exercise within one semester, say 45 hours.

In this paper, we will introduce the pipelined Aizup model and experiences in designing the circuits with Xilinx XC4000 library. The paper is organized as follows. Section 2 introduces the pipelined processor architecture model. Sections 3 and 4 discuss circuit design and functional simulation, respectively. Section 5 is dedicated to the Aizup implementation on FPGA chip. The final section concludes the circuit design and the Xilinx design tools that cooperate with Cadence.

2 Processor Architecture Model

The aim of the exercise for Computer Architecture/Organization Education is to have students understand well the hardware structure and the operation of pipelined processor and master the circuit design methodology. In order to be able to implement the pipelined processor associated with instruction memory and data memory on a single FPGA chip, we select the 8 bits as the width of instructions and data. Most of the instructions have two operands, one is a register operand (RD) and the other is a register operand (RS) or an immediate. The operation result is written back to RD. The Aizup implements four pipeline stages, they are IF (Instruction Fetch), DC (DeCode and operand fetch), EX (EXecution or memory access), and WB (Write Back).

Table 1 lists the instruction operation codes, the operations in each stage, and the immediate extensions. Some registers are needed for the pipeline operations. They are PC (Program Counter), IR (In-

Table 1: Instruction set and pipeline operations

Instruction Format			Operations	IF	DC	EX	WR
NOP 0000	 00	 00	No Operation	IR←M[PC] PC←PC+1			
ADD 0001	RD, RD	RS RS	RD←RD + RS Update Z	IR←M[PC] PC←PC+1	A←RD B←RS	C←A + B Z←((A + B)==0)	RD←C
SUB 0010	RD, RD	RS RS	RD←RD − RS Update Z	IR←M[PC] PC←PC+1	A←RD B←RS	C←A − B Z←((A − B)==0)	RD←C
OR 0011	RD, RD	RS RS	RD←RD or RS Update Z	IR←M[PC] PC←PC+1	A←RD B←RS	C←A or B Z←((A or B)==0)	RD←C
AND 0100	RD, RD	RS RS	RD←RD and RS Update Z	IR←M[PC] PC←PC+1	A←RD B←RS	C←A and B Z←((A and B)==0)	RD←C
XOR 0101	RD, RD	RS RS	RD←RD xor RS Update Z	IR←M[PC] PC←PC+1	A←RD B←RS	C←A xor B Z←((A xor B)==0)	RD←C
MOV 0110	RD, RD	RS RS	RD←RS Update Z	IR←M[PC] PC←PC+1	A←RD B←RS	C←B Z←(B==0)	RD←C
LD 0111	RD, RD	RS RS	RD←M[RS]	IR←M[PC] PC←PC+1	A←RD B←RS	C←M[B]	RD←C
ST 1000	RD, RD	RS RS	M[RS]←RD	IR←M[PC] PC←PC+1	A←RD B←RS	M[B]←A	
ADDI 1001	RD, RD	n n	RD←RD + 000000n Update Z	IR←M[PC] PC←PC+1	A←RD B←000000n	C←A + B Z←((A + B)==0)	RD←C
SUBI 1010	RD, RD	n n	RD←RD − 000000n Update Z	IR←M[PC] PC←PC+1	A←RD B←000000n	C←A − B Z←((A − B)==0)	RD←C
SR0L 1011	N N		R0←R0 or 0000N Update Z	IR←M[PC] PC←PC+1	A←R0 B←0000N	C←A or B Z←((A or B)==0)	R0←C
SR0H 1100	N N		R0←N0000 Update Z	IR←M[PC] PC←PC+1	A←R0 B←N0000	C←B Z←(B==0)	R0←C
BZ 1101	N N		if (Z) PC←PC+(s)N	IR←M[PC] PC←PC+1	if (Z) PC←PC+(s)N		
BNZ 1110	N N		if (!Z) PC←PC+(s)N	IR←M[PC] PC←PC+1	if (!Z) PC←PC+(s)N		
BRA 1111	N N		PC←PC+(s)N	IR←M[PC] PC←PC+1	PC←PC+(s)N		

struction Register), register A and B (holding two source operands), and register C (holding the result of operations). There is also a single bit register, Z, for storing the zero tag, that will be evaluated by conditional branch instructions.

For the data dependency, consider the following sequence of pipeline executions of I1, I2, and I3. The pipeline operations are shown in Figure 1.

```
I1:  ADD  R1, R2 ; R1 <- R1 + R2
I2:  SUB  R3, R1 ; R3 <- R3 - R1
I3:  SUBI R1, 1  ; R1 <- R1 - 1
```

The instruction I2 reads R3 and R1 from register file in DC stage, and writes them to A and B respectively. At the same time, the instruction I1 calculates the sum of R1 and R2 and write it to C. I1 will write the sum to R1 in the next stage. Therefore, I2 will get old content from register file and write it to B. If I2 uses it for subtraction, then a wrong result will be generated. There is no problem with I3 reading R1.

A NOP instruction can be inserted between I1 and I2 to delay the execution of I2, but the performance will suffer. We can design a special data path to solve this problem. A dependency detection block can detect the dependencies. When a data dependency is detected, the source data for ALU operation is passed from C via multiplexor, instead of from A or B.

The data dependencies (ADEPEN/BDEPEN) will

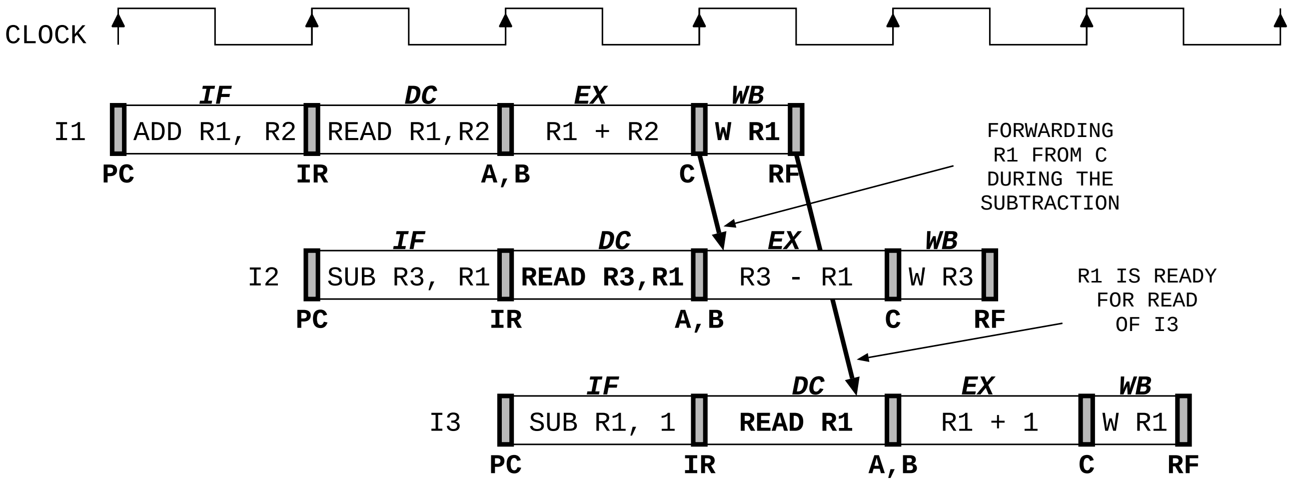


Figure 1: Data dependency and data forwarding

happen if

1. the operand A/B of the current instruction is a register operand (cRD/cRS),
2. the result of the previous instruction will be written into register file (pRD), and
3. cRD/cRS and pRD are the same register.

The dependency detection takes place in EX stage. In our processor design, we move it to DC stage, and use pipeline registers to transfer to EX stage. There are good points of the movement. First, doing the detection in DC stage will save the gates because some common logic can be shared with other decode circuits. Second, the time required by EX stage will be shortened because the ADEPEN and BDEPEN are available immediately at the beginning of EX stage.

As for the control dependency, we adopt a delay branch method. In our processor model, the branch target address is evaluated in DC stage. It means that one delay cycle is introduced and an additional adder is needed for address evaluation. We can use this method to demonstrate the optimization by reorganizing the instruction codes.

3 Processor Design with Xilinx XC4000 Library

Figure 2 shows the organization of Aizup. The major cells include ALU, register file, adders, instruction memory, data memory, decoder, pipeline registers, and multiplexors. The four pipeline stages are marked in the left side of the figure.

3.1 ALU Design

For implementing the Aizup instruction set, six operations should be done by ALU. The six operations are (1) PassB (for MOV and SR0H), (2) OR (for OR and SR0L), (3) XOR (for XOR), (4) AND (for AND), (5) SUB (for SUB and SUBI), and (6) ADD (for ADD and ADDI). A 3-bit signal (AOP<2:0>) selects the ALU operation.

3.2 Register File Design

In Aizup, the register-to-register instructions perform RD←RD *op* RS, and two bits are used for addressing the register file. It means a 4×8 bits register file with two read ports and a write port is needed. It can be implemented with four 8-bit registers plus a pair of 4-to-1 multiplexors each 8 bits wide for read ports and a 2-to-4 decoder for write control. The circuitry of register file is shown in Figure 3.

3.3 Data Memory Design

In Aizup, there are separated instruction memory module and data memory module. Each has 256x8 bits (8 bits address bus and 8 bits data bus). There is no external memory chip mounted in the evaluation board, therefore we use Xilinx ram32x8 modules

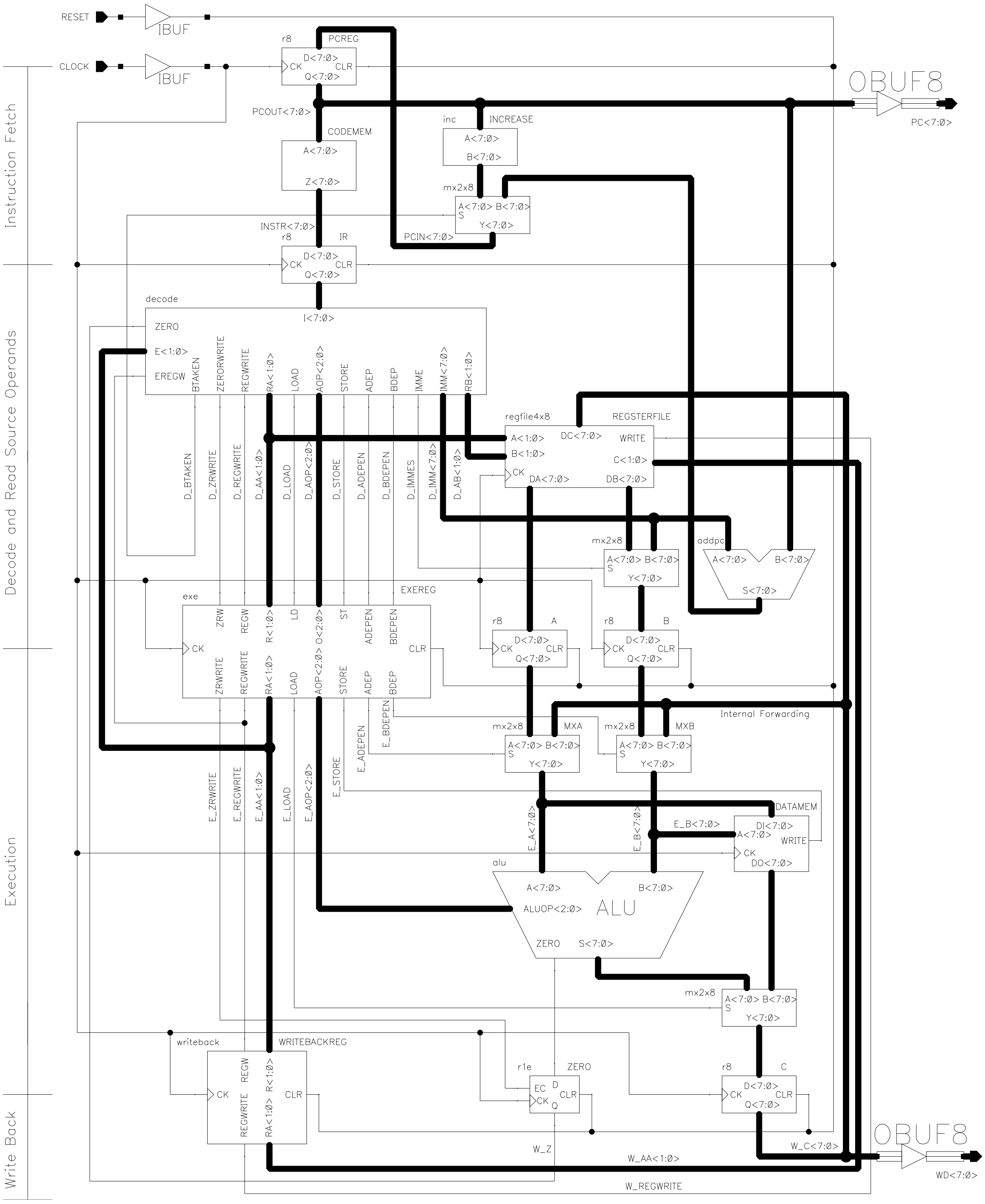


Figure 2: The top schematic of Aizup

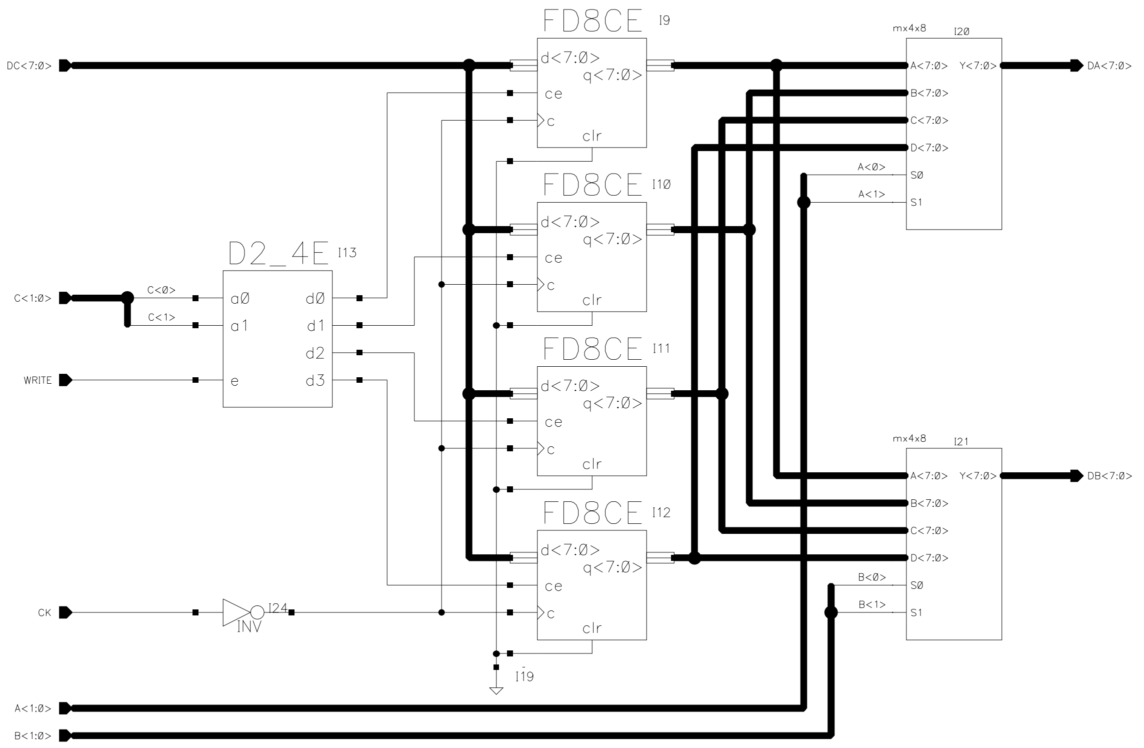


Figure 3: 4 × 8 bits register file

to build on-chip memory. It becomes possible to access memory once at each cycle. Comparing to register access, the memory access control circuit is more complicated. Especially, in the memory write operation (WE), the glitches on the WE line or glitches on the address lines while WE is active might cause problems [1]. An example of a glitch-free WE generation circuit is given in [1], but the memory access will take more than one clock cycle. The performance will suffer.

Here we show a glitch-free WE and address generation circuit that can perform memory access once on every clock cycle. An example of 32x8 data memory module is shown in Figure 4. In order to match the XC4000 RAM in the design, a FD8CE is added. It postpones the data half cycle. And also a latch (ld5) is used for latching the address. The timing chart of write-cycle is shown in Figure 5.

3.4 Instruction Memory Design

For the instruction memory design, we use Xilinx ROM32x1 components to build an on-chip instruction memory module. The ROM must be initialized to a known value with the INIT = value parameter. The following code segment is used for testing the Aizup.

```
ADR INST-CODE  INSTRUCTION   COMMENT
00: 1100 00 00 SROH 0      ; R0 = 0
01: 1011 01 00 SROL 4      ; R0 = 4
02: 0110 01 00 MOV  R1, R0 ; R1 = 4, COUNTER
03: 1100 00 00 SROH 0      ; R0 = 0
04: 0110 10 00 MOV  R2, R0 ; R2 = MEMORY ADDRESS
05: 0110 11 00 MOV  R3, R0 ; R3 = RESULT DATA
L0000:
06: 1001 11 10 ADDI R3, 2  ; R3 = 2, 4, 6, 8
07: 1000 11 10 ST   R3, R2 ; STORE TO MEM 0,1,2,3
08: 1001 10 01 ADDI R2, 1  ; ADDRESS + 1
09: 1010 01 01 SUBI R1, 1  ; COUNTER - 1
0A: 0000 00 00 NOP         ; WAIT FOR ZERO TAG
0B: 1110 10 10 BNZ  L0000  ; CONT IF NOT FINISHED
0C: 0000 00 00 NOP         ; DELAY INST, EXECUTED
0D: 1100 00 00 SROH 0      ; R0 = 0
0E: 1011 01 00 SROL 4      ; R0 = 4
0F: 0110 01 00 MOV  R1, R0 ; R1 = 4, COUNTER
10: 1100 00 00 SROH 0      ; R0 = 0
11: 0110 10 00 MOV  R2, R0 ; R2 = MEMORY ADDRESS
```

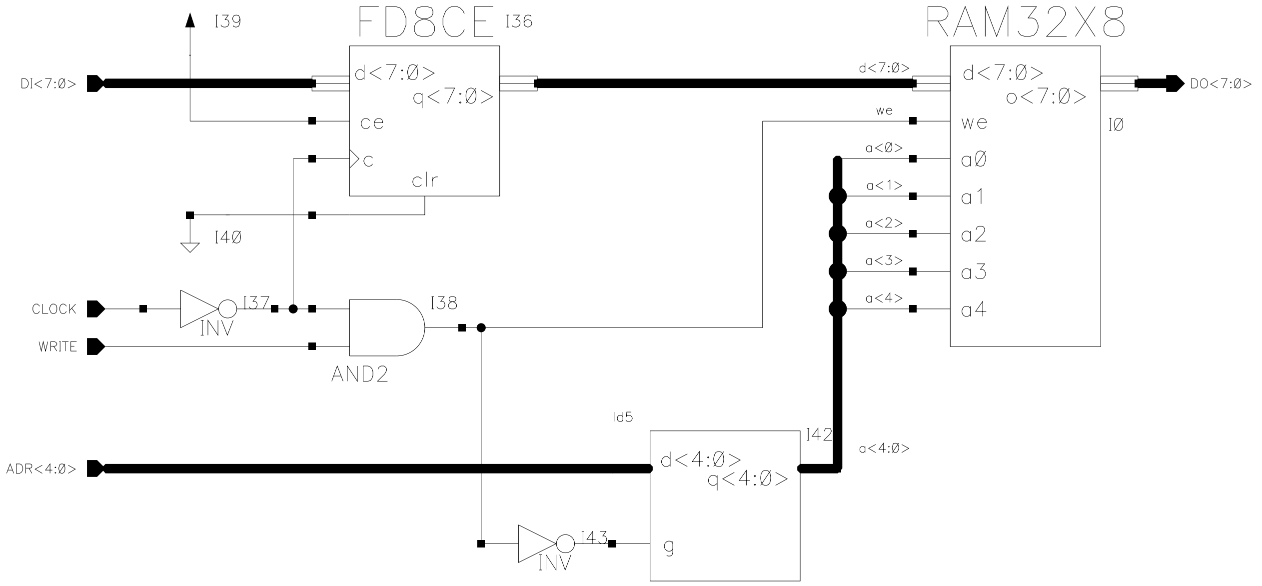


Figure 4: A glitch-free WE and address generation circuit

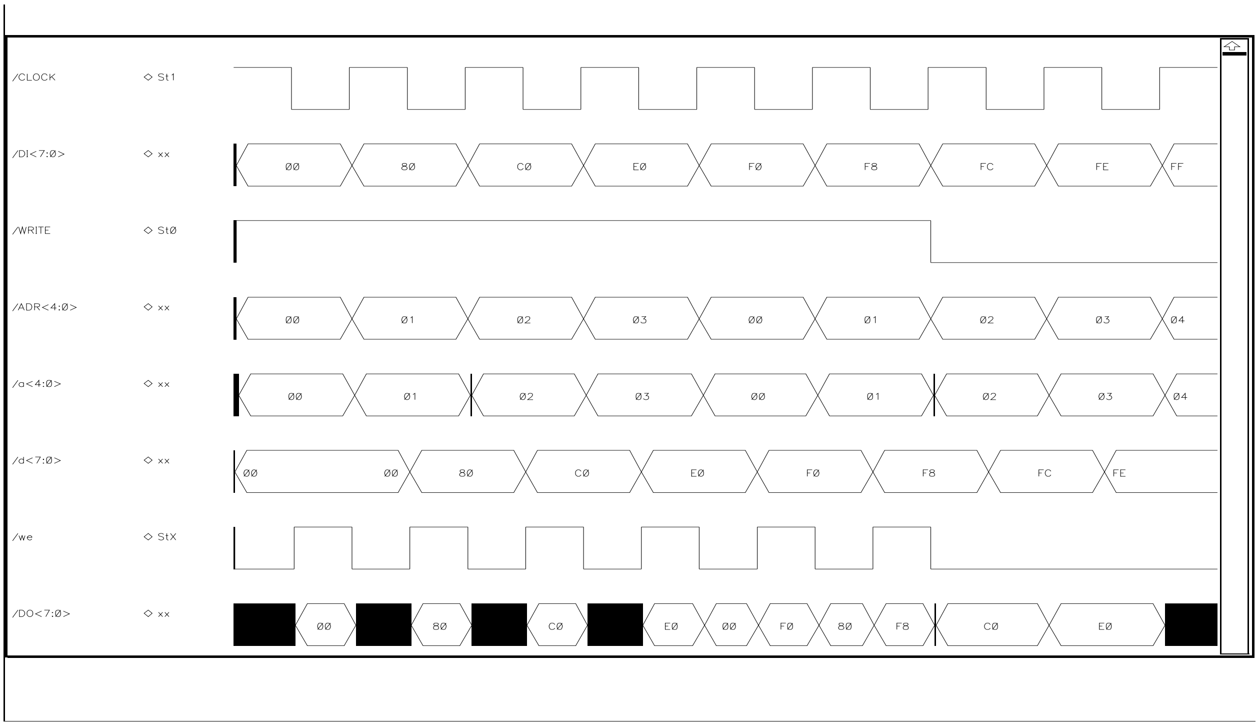
```
L0001:
12: 0111 11 10 LD   R3, R2 ; R3 = LOADED DATA
13: 0001 00 11 ADD  R0, R3 ; R0 = R0 + R3
14: 1001 10 01 ADDI R2, 1  ; ADDRESS + 1
15: 1010 01 01 SUBI R1, 1  ; COUNTER - 1
16: 0000 00 00 NOP         ; WAIT FOR ZERO TAG
17: 1110 10 10 BNZ  L0001  ; CONT IF NOT FINISHED
18: 0000 00 00 NOP         ; DELAY INST, EXECUTED
19: 0110 11 00 MOV  R3, R0 ; R3 = RESULT (SUM)
1A: 1100 00 00 SR0H 0      ; R0 = 0
1B: 1011 01 00 SR0L 4      ; R0 = 4
1C: 1000 11 00 ST   R3, R0 ; STORE RESULT TO MEM 4
1D: 0000 00 00 NOP         ;
1E: 0000 00 00 NOP         ;
1F: 0000 00 00 NOP         ;
```

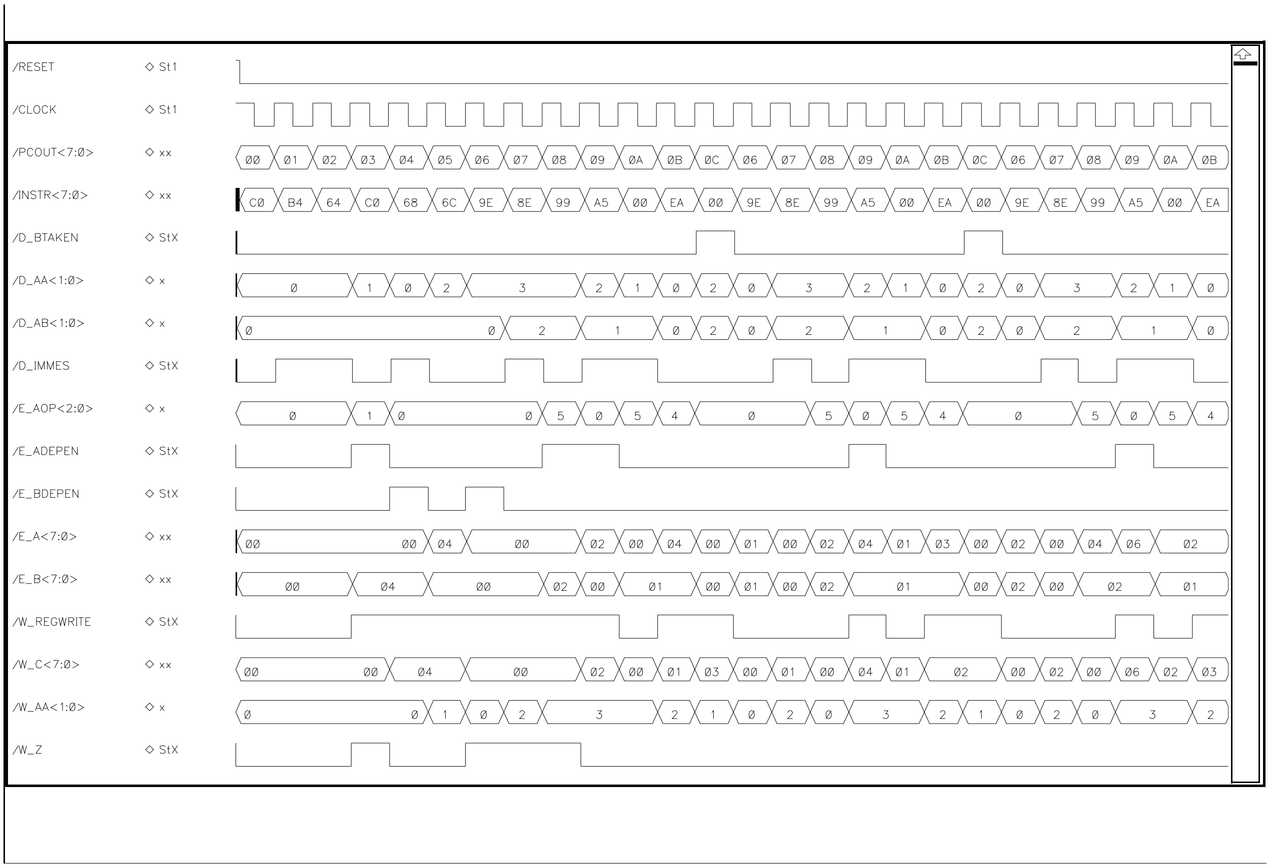
The NOP instructions are used for waiting zero tag and for the delay branch mechanism. We can ask students to reorganize the instructions to remove the NOP instructions.

3.5 Control Unit Design

The control unit generates all control signals for controlling datapath. We summarize the control signals in Aizup as following.

1. IF stage
    - BTAKEN (Branch Taken). If branch is taken, BTAKEN should be high to select the branch target address for instruction fetching in the next clock cycle.
2. DC stage
    - AA<1:0> (A Address). It indicates the register number of RD. RD is the destination register but it is also the source 1 register (RD←RD *op* RS). Most of the instructions put the number of RD in a fixed position in the instruction format, but the SR0L and SR0H instructions are special cases. The number of RD is always 0 for these instructions.
    - AB<1:0> (B Address). It indicates the register number of RS. RS is the source 2 register. Most of the instructions put the number of RS in a fixed position in the instruction format, but some instructions put immediate in that position.
    - IMMES (Immediate Selection). When the source 2 operand is an immediate, IMMES should be high for selecting the immediate, not register operand.
    - IMM<7:0> (Immediate). It is an 8-bit immediate. Based on instructions, it can be generated with different extension methods. The IMM<7:0> is used not only for the


/CLOCK ◇ St1
/DI<7:0> ◇ xx
00 80 C0 E0 F0 F8 FC FE FF
/WRITE ◇ St0
/ADR<4:0> ◇ xx
00 01 02 03 00 01 02 03 04
/a<4:0> ◇ xx
00 01 02 03 00 01 02 03 04
/d<7:0> ◇ xx
00 00 80 C0 E0 F0 F8 FC FE
/we ◇ StX
/DO<7:0> ◇ xx
00 80 C0 E0 00 F0 80 F8 C0 E0


/RESET ◇ St1
/CLOCK ◇ St1
/PCOUT<7:0> ◇ xx 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 06 07 08 09 0A 0B 0C 06 07 08 09 0A 0B
/INSTR<7:0> ◇ xx C0 B4 64 C0 68 6C 9E 8E 99 A5 00 EA 00 9E 8E 99 A5 00 EA 00 9E 8E 99 A5 00 EA
/D_BTAKEN ◇ StX
/D_AA<1:0> ◇ x 0 1 0 2 3 2 1 0 2 0 3 2 1 0 2 0 3 2 1 0
/D_AB<1:0> ◇ x 0 0 2 1 0 2 0 2 1 0 2 0 2 1 0
/D_IMMES ◇ StX
/E_AOP<2:0> ◇ x 0 1 0 0 5 0 5 4 0 5 0 5 4 0 5 0 5 4
/E_ADEPEN ◇ StX
/E_BDEPEN ◇ StX
/E_A<7:0> ◇ xx 00 00 04 00 02 00 04 00 01 00 02 04 01 03 00 02 00 04 06 02
/E_B<7:0> ◇ xx 00 04 00 02 00 01 00 01 00 02 01 00 02 00 02 01
/W_REGWRITE ◇ StX
/W_C<7:0> ◇ xx 00 00 04 00 02 00 01 03 00 01 00 04 01 02 00 02 00 06 02 03
/W_AA<1:0> ◇ x 0 0 1 0 2 3 2 1 0 2 0 3 2 1 0 2 0 3 2
/W_Z ◇ StX

to 224 bytes. The result is like below.

```
Preliminary estimate of device utilization for part  4006PC84:
-------------------------------------------------------------
  30% utilization of I/O pins.                  ( 18 of  61)
  98% utilization of CLB function generators.   (503 of 512)
  19% utilization of CLB flip-flops.            ( 95 of 512)
-------------------------------------------------------------
```

By using Back-Annotation function, we can get a pld_schematic that is the very-top schematic of our circuitry with the chip as a component. We can see the pins' assignment. We can also do the functional simulation on it. The contents of ROM can be read out in the simulation.

The bitstream file is transformed to Intel mcs format (aizup.mcs), and then it is downloaded to the evaluation board via RS-232C. After we program the XC4006PC84 chip, we can use Logic Analyzer to test the output pins and analyze the timing diagram.

6 Conclusions

In this paper, we presented a pipelined processor design and implementation on Xilinx XC4006PC84 for the exercise of Computer Architecture / Organization Education at the University of Aizu. Through whole the procedure, the undergraduate students can understand well the hardware circuits of the pipelined processors, and more important, they can master the design methodologies by using CAD/CAE design tools.

The Aizup model that we designed and implemented does not contain any floating-point instruction. We have designed a Multiple-threaded Multiple-pipelined processor including floating-point unit using Toshiba TC180/183E/C ASIC library [7] [8]. The floating-point unit can perform addition, subtraction, multiplication, division, square root, and conversion between integer and floating-point numbers. We are now investigating the possibility of implementation of the floating-point unit on FPGA chips for advanced exercise course.

We are also investigating the use of external memory modules so that we can demonstrate the external memory access and instruction/data cache operations. Multiple FPGA chips will be mounted in the board so that we can realize multiprocessor design on the board.

As for the Xilinx design tools that cooperate with Cadence, there are some restrictions, to which we have to pay attention. First, it is not allowed to change the instance name to realize an array of instances. For example, we cannot change the name, I1 for example, of or2 component to I1<7:0> for or2x8. We have to put eight individually. Sometime we feel this is good for view, but sometime we feel it is not convenient to use. The second problem of the Xilinx design tools is that it can not recognize a cell that is named with capital characters. The third is that when the netlisting is carried out, it cannot automatically create a directory for a cell that has a multi-sheet structure. Users have to create a directory manually using the same name of the cell before starting the netlisting. The last problem we encountered is that the initialized ROM cannot be read out for functional simulation before implementation.

References

[1] R. Iwanczuk, "Using the XC4000 RAM Capability," *The Programmable Logic Data Book.* Xilinx, 1994.

[2] J. Hennessy and D. Patterson, Computer Architecture, A Quantitative Approach, Morgan Kaufmann Publishers, Inc., 1990.

[3] D. Patterson and J. Hennessy, Computer Organization & Design: The Hardware/Software Interface, Morgan Kaufmann Publishers, Inc., 1994.

[4] XACT Development System - XACT User Guide, Xilinx, April, 1994.

[5] XACT Development System - XACT Libraries Guide, Xilinx, April, 1994.

[6] Y. Li and W. Chu, "Aizup – A Pipelined Processor Design and Its Implementation on XILINX FPGA Chip," *Lecture Notes*, The University of Aizu, 1995.

[7] Y. Li and W. Chu, "Design and Performance Analysis of A Multiple-threaded Multiple-pipelined Architecture," *Proc. of the Second International Conference on High Performance Computing*, New Delhi, India, December 1995.

[8] Y. Li and W. Chu, "A New Non-Restoring Square Root Algorithm and Its VLSI Implementations," 1996, to be submitted.

[9] Y. Li and W. Chu, "Using Computer Architecture/Organization at the University of Aizu," *Second Annual Workshop on Computer Architecture Education*, San Jose, California, February, 1996.