CSCE 212 001 Introduction to Computer Architecture
Project, Due 05/02/2019, Last change: 04/22

A CPU Simulator

In this project, you will write a program to simulate the essential part of a CPU to execute a program. The project includes three major components, an instruction assembler and decoder, the CPU simulator itself that executes instructions in five stages (IF, ID, EXE, MEM and WB), and a cache simulator. Your final CPU simulator should be able to execute the provided test program and produce the expected results. The three components of the project will first be implemented in Homework #4, #5 and #6 and at the end, you will need to integrate them into one program. While C and Linux are the preferred language and OS for the implementation, you can choose Java or C++, and other OS (Windows or Mac OS X) that you are comfortable with to work on. But the template code you are given to start with is in C only. If you need to choose other languages than C, C++ or Java, please confirm with me first.

The CPU supports the following 7 instructions:

- R-Type arithmetic/logical instruction: add, and sub
- I-Type memory reference: lw, sw
- I-Type control transfer: beq, j
- I-Type add immediate: addi

1. Assembler, Disassembler and Decoder:

Instruction format and decoding

An instruction is encoded as a 32-bit word, [31:0]. We use very similar encoding schema as MIPS. 6 bits ([31:26]) are used for encoding the function of an instruction, according to the following table.

Instructions	[31:26] bits	Decimal number
ADD	000000	0
SUB	000001	1
LWR	000010	2
	000011	3
	000100	4
ADDI	000101	5
	000110	6
	000111	7
LW	001000	8
SW	001001	9
	001010	10
	001011	11
BEQ	001100	12
	001101	13
	001110	14
J	001111	15

R-type ALU instructions (add and sub), used in the source code as "Ins, Rd, Rs, Rt" format, are encoded as instruction words of [func][Rs][Rt][Rd][unused] format. Please note the order of the three operands in the instruction word is different from the order of the operands in the source code. The func field is [31:26] as mentioned before. Rs, Rt and Rd operands, each with 5 bits, are encoded in the following 15

bits, Rs: [25:21], Rt: [20:16], and Rd: [15:11]. So of a 32-bit instruction word, a R-type instruction only uses 6+15 = 21 bits. The rest 11 bits are not used.

The LW/SW/Beq/Addi I-type instructions, used in the source code as "LW|SW|BEQ|ADDI, Rt, Rs, imm", are encoded as instruction words of [func][Rs][Rt][Imm]. Please note the order of the Rt and Rs operands in the instruction word is different from the order of the instruction in the source code. The Rs and Rt operands, each with 5 bits, are encoded as Rs: [25:21] and Rt: [20:16]. The rest 16 bits [15:0] are used for encoding the immediate.

The J instruction, used in the source code as "J absolute-target", is encoded in an instruction word of [func][absolute-target]. The absolute-target is an immediate encoded in bit [25:0].

You should start from the provided assembler_decoder.c file for your implementation. It already includes the support for ADD instruction. You need to read the code and understand the details of how ADD is implemented in the assembler, decoder and disassembler. Your implementation should finish the code in those TODO places to support all the other instructions. While C and Linux are the preferred language and OS for the implementation, you can choose Java or C++, and other OS (Windows or Mac OS X) that you are comfortable with to work on. But the template code you are given to start with is in C only. If you need to choose other languages than C, C++ or Java, please confirm with me first. A test program, test.asm, is provided for testing your implementation. Currently if you compile and run the program with test.asm as following, its output shows that only ADD instruction can be assembled, decoded and disassembled. For other instructions, "Unrecognized instruction …" message is printed out.

```
yanyh@vm:~/csce212_simcpu$ gcc assembler_decoder.c -o assembler_decoder
yanyh@vm:~/csce212_simcpu$ ./assembler_decoder test.asm
ADDI, $s3, $s0, 1           # instruction #0, i = 1;
        Unrecognized instruction: ADDI, ignore.
ADDI, $s4, $s0, 256         # instruction #1, Init N=256
        Unrecognized instruction: ADDI, ignore.
ADDI, $s4, $s4, -2          # instruction #2, $s4 has 255
        Unrecognized instruction: ADDI, ignore.
BEQ, $s3, $s4, 12           # 3, Jump to the end of the code, use relative address 12
        Unrecognized instruction: BEQ, ignore.
ADD, $s11, $s3, $s3         # 4,  i = i*2
        0x00635800: ADD, $s11, $s3, $s3
ADD, $s11, $s11, $s11       # 5,  i = i*2*2, now $s11 has i*4
        0x016b5800: ADD, $s11, $s11, $s11
ADD, $s5, $s11, $s2         # 6,  &B[i] is now in $s5
        0x01622800: ADD, $s5, $s11, $s2
LW,  $s6, $s5, -4           # 7,  B[i-1] is now in $s6
        Unrecognized instruction: LW, ignore.
LW,  $s7, $s5, 0            # 8,  B[i] is now in $s7
        Unrecognized instruction: LW, ignore.
LW,  $s8, $s5, 4            # 9,  B[i+1] is now in $s8
        Unrecognized instruction: LW, ignore.
ADD, $s9, $s6, $s7          # 10, B[i-1] + B[i]
        0x00c74800: ADD, $s9, $s6, $s7
ADD, $s9, $s8, $s9          # 11, B[i-1] + B[i] + B[i+1]
        0x01094800: ADD, $s9, $s8, $s9
ADD, $s10, $s11, $s1        # 12, &A[i] is now in $s10
        0x01615000: ADD, $s10, $s11, $s1
SW,  $s9, $s10, 0           # 13, A[i] stored the result
        Unrecognized instruction: SW, ignore.
ADDI, $s3, $s3, 1           # 14, i++
        Unrecognized instruction: ADDI, ignore.
J, 3                        # 15, Jump instruction #3 (BEQ instruction, use absolute address)
        Unrecognized instruction: J, ignore.
```

After your implementation, your program should output the same or similar output as in the following screenshot using the test.asm file as input. Your program will be evaluated using another example program when being graded. Your submission needs to include your source code and the screenshot of your execution (similar to mine).

```
yanyh@vm:~/csce212_simcpu$ ./assembler_decoder test.asm
ADDI, $s3, $s0, 1            # instruction #0, i = 1;
        0x14030001: ADDI, $s3, $s0, 1
ADDI, $s4, $s0, 256          # instruction #1, Init N=256
        0x14040100: ADDI, $s4, $s0, 256
ADDI, $s4, $s4, -2           # instruction #2, $s4 has 255
        0x1484fffe: ADDI, $s4, $s4, -2
BEQ, $s3, $s4, 12            # 3, Jump to the end of the code, use relative address 12
        0x3083000c: BEQ, $s3, $s4, 12
ADD, $s11, $s3, $s3          # 4,  i = i*2
        0x00635800: ADD, $s11, $s3, $s3
ADD, $s11, $s11, $s11        # 5,  i = i*2*2, now $s11 has i*4
        0x016b5800: ADD, $s11, $s11, $s11
ADD, $s5, $s11, $s2          # 6,  &B[i] is now in $s5
        0x01622800: ADD, $s5, $s11, $s2
LW,  $s6, $s5, -4            # 7,  B[i-1] is now in $s6
        0x20a6fffc: LW, $s6, $s5, -4
LW,  $s7, $s5, 0             # 8,  B[i] is now in $s7
        0x20a70000: LW, $s7, $s5, 0
LW,  $s8, $s5, 4             # 9,  B[i+1] is now in $s8
        0x20a80004: LW, $s8, $s5, 4
ADD, $s9, $s6, $s7           # 10, B[i-1] + B[i]
        0x00c74800: ADD, $s9, $s6, $s7
ADD, $s9, $s8, $s9           # 11, B[i-1] + B[i] + B[i+1]
        0x01094800: ADD, $s9, $s8, $s9
ADD, $s10, $s11, $s1         # 12, &A[i] is now in $s10
        0x01615000: ADD, $s10, $s11, $s1
SW,  $s9, $s10, 0            # 13, A[i] stored the result
        0x25490000: SW, $s9, $s10, 0
ADDI, $s3, $s3, 1            # 14, i++
        0x14630001: ADDI, $s3, $s3, 1
J, 3                         # 15, Jump instruction #3 (BEQ instruction, use absolute address)
        0x3c000003: J, 3
```

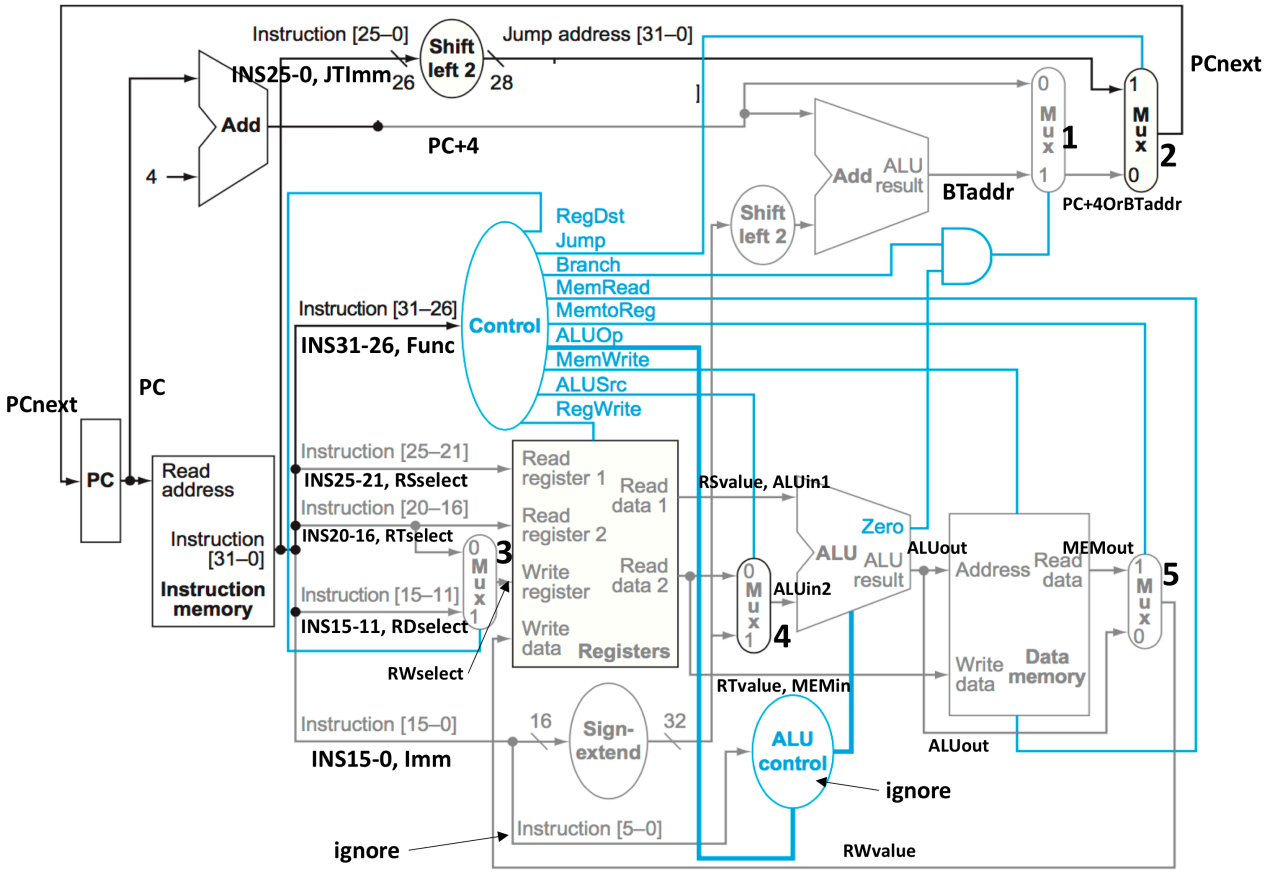

FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at

2. CPU Functional Simulator

The second component of the project is to create a functional simulator of CPU of the above diagram using a program. In the program, CPU components (instruction memory, registers, ALU, and data memory), datapath, control signal and logic in the CPU diagram are implemented in the source code to support the designed 7 instructions. Question 1 and 2 of Homework 5 give you some exercises on how data are moved on the datapath and how control signals are set and used by the CPU when an instruction is being executed. Your program implements those data movement and logics in the program, simulating a real CPU execution. The initial part of program is already provided to you, cpusim.c. The program includes most of the implementation of the simulator and you need to add the code in the TODO places in the program:

Your tasks:

Read the code and comments first to understand the overall structure of the simulator code and then fill in code in those TODO places to complete the simulator for the 7 instructions. When you add code in those TODO places, you should refer to the CPU diagram for adding statements for setting datapath and control.

Execute the simulator program with provided input and collect trace data:

The test.asm program and its binary test.asm.bin are provided for you to test your implementation. The test.asm.bin program is the binary program created from the test.asm file by the assembler. The test.asm is different from the file in the homework 4 since we introduced addi instruction. The assembler program is also revised and provided to you so you can assembly the test.asm program to binary

program (test.asm.bin) if you need to make changes of the test.asm file. You are welcome to revise your own assembler, but not required. If you do not plan to change the test.asm file, you can just use the test.asm.bin file provided to test your CPU simulator.

When you are working on your simulator program (cpusim.c), you should run with test.asm.bin as input. The program will run and save the program execution into a trace file named "cpusim_trace.txt". Without changing anything of the cpusim.c file, if you compile it and execute the program with test.asm.bin input, it will output "Verification Failed" message and the trace will contain more information as follows. (cat is the command to list the content of a file.)

```
yanyh@vm:~/csce212_simcpu$ gcc cpusim.c -o cpusim
yanyh@vm:~/csce212_simcpu$ ./cpusim test.asm.bin
Verification Failed!
yanyh@vm:~/csce212_simcpu$ cat cpusim_trace.txt
Fetch instruction 14030001 at PC 0
      Decode instruction (fun rs rt rd Imm JTImm): ADDI 0 3 0 1 196609
      Fetch register: Rs: Reg[0]=0, Rt: Reg[3]=0
      EXE: Ops ADD, ALUout: 0, Zero: 0, BTaddr: 0
      MEM: PCnext: 0
Simulation goes to infinite loop of test.asm program, terminate it
Verification failed: VA[1]: -850272457, Sim Number: 0
Verification failed: VA[2]: 716293359, Sim Number: 0
Verification failed: VA[3]: 1122055173, Sim Number: 0
Verification failed: VA[4]: -788691734, Sim Number: 0
Verification failed: VA[5]: -1589852331, Sim Number: 0
Verification failed: VA[6]: 1805886067, Sim Number: 0
Verification failed: VA[7]: -1511266276, Sim Number: 0
Verification failed: VA[8]: -1247878484, Sim Number: 0
Verification failed: VA[9]: -1159732566, Sim Number: 0
Verification failed: VA[10]: -576257088, Sim Number: 0
Verification failed: VA[11]: -718276064, Sim Number: 0
Verification failed: VA[12]: 2650504, Sim Number: 0
Verification failed: VA[13]: -348487530, Sim Number: 0
```

After you make changes and you should compile and run the program with test.asm.bin as input. If you do it correctly, the program output and content of the trace file will be as follows:

```
yanyh@vm:~/csce212_simcpu$ gcc cpusim.c -o cpusim
yanyh@vm:~/csce212_simcpu$ ./cpusim test.asm.bin
Simulation and Verification Passed Successfully!
yanyh@vm:~/csce212_simcpu$ less cpusim_trace.txt
Fetch instruction 14030001 at PC 0
        Decode instruction (fun rs rt rd Imm JTImm): ADDI 0 3 0 1 196609
        Fetch register: Rs: Reg[0]=0, Rt: Reg[3]=0
        EXE: Ops ADD, ALUout: 1, Zero: 0, BTaddr: 8
        MEM: PCnext: 4
        WB: Reg[3] = 1
Fetch instruction 14040010 at PC 4
        Decode instruction (fun rs rt rd Imm JTImm): ADDI 0 4 0 16 262160
        Fetch register: Rs: Reg[0]=0, Rt: Reg[4]=0
        EXE: Ops ADD, ALUout: 16, Zero: 0, BTaddr: 72
        MEM: PCnext: 8
        WB: Reg[4] = 16
```

……

```
Fetch instruction 3083000c at PC 12
        Decode instruction (fun rs rt rd Imm JTImm): BEQ 4 3 0 12 8585228
        Fetch register: Rs: Reg[4]=14, Rt: Reg[3]=14
        EXE: Ops SUB, ALUout: 0, Zero: 1, BTaddr: 64
        MEM: PCnext: 64
Fetch instruction 3c0f423f at PC 64
        Decode instruction (fun rs rt rd Imm JTImm): J 0 15 8 16959 999999
        Fetch register: Rs: Reg[0]=0, Rt: Reg[15]=0
        EXE: Ops ADD, ALUout: 0, Zero: 1, BTaddr: 67904
        MEM: PCnext: 3999996
=================================================
Simulation and Verification Passed Successfully!
Simulation Summary:
         Num of Instruction Executed: 174
(END)
```

Your submission should include both the trace file and the cpusim.c source code. See below ("less" command is used to view the trace file).

If you need to change the problem size of the program in test.asm, you need to modify the following line and change 16 to a different number.

```
ADDI, $s4, $s0, 16           # instruction #1, Init N=16
```

Because of the way we implement the verification of the CPU simulator, you will then need to change the N value in the cpusim.c file to have the same number as you put in the test.asm file.

```
int N = 16;             /* This has to be exactly the same as the number in the
                 * line "ADDI, $s4, $s0, 16  # instruction #1, Init N=16" of the
test.asm source code */
```

After that, you will need to assembly the changed test.asm file to create a new binary program. You also need to recompile the cpusim.c and feed it with the new test.asm.bin file. Below is the command sequence to make the changes of test.asm and cpusim.c, and to run the simulation. (command outputs are not shown and you can use other editor (not necessarily vi) to make changes of the files).

```
yanyh@vm:~/csce212_simcpu$ vi test.asm
yanyh@vm:~/csce212_simcpu$ ./assembler_decoder test.asm
yanyh@vm:~/csce212_simcpu$ ls test.asm.bin
test.asm.bin
yanyh@vm:~/csce212_simcpu$ vi cpusim.c
yanyh@vm:~/csce212_simcpu$ gcc cpusim.c  -o cpusim
yanyh@vm:~/csce212_simcpu$ ./cpusim test.asm.bin
```

3. CPU Simulator with Cache Simulator

The last component of the project is to add cache simulation to your CPU functional simulator you implement. Cache simulation should support both instruction cache (I-Cache) and data cache (D-Cache). To simply the design and verification, I-Cache has 4 2-word blocks and D-Cache has 64 4-word blocks. They both are directed-mapped and write-through and write-allocate policies should be used for writing a word to the D-Cache and data memory. The initial part of program is already provided to you, cpusim_cachesim.c. The program and the new test program (test256.asm.bin) are both updated for computing 256 element 1-D stencil.

Tasks and procedure:

The cpusim_cachesim.c program includes most of the implementation and you need to add the code in the TODO places in the program.

1. Download cpusim_cachesim.c file, compile and execute with test256.asm.bin as input. It should show the following output since it has not yet completed.

```
yanyh@vm:~/csce212_simcpu$ gcc cpusim_cachesim.c -o cpusim_cachesim
yanyh@vm:~/csce212_simcpu$ ./cpusim_cachesim test256.asm.bin
Verification Failed!
yanyh@vm:~/csce212_simcpu$
```

You can check the cpusim_trace.txt file using "`cat cpusim_trace.txt`" command as you did for the second component of the project.

2. Migrate (copy) your implementation of those TODO places from your cpusim.c file to the cpusim_cachesim.c file. After that, compile and run your program with the test256.asm.bin file. You program should complete with the successful message, i.e. "`Simulation and Verification Passed Successfully!`" if you implementation of cpusim.c is correct and you migrate them correctly.

```
yanyh@vm:~/csce212_simcpu$ gcc cpusim_cachesim.c -o cpusim_cachesim
yanyh@vm:~/csce212_simcpu$ ./cpusim_cachesim test256.asm.bin
Simulation and Verification Passed Successfully!
```

3. Read the `FetchInstructionWord` function (line 174) to understand how I-Cache simulation is implemented. You will also need to read code for the struct defined to support I-Cache and D-Cache. They are from line 127 to 160 in the cpusim_cachesim.c file, including the definition of the **`struct`** `AddressToICacheAccess`, **`struct`** `InstructionCacheEntry`, **`struct`** `AddressToDCacheAccess`, **`struct`** `DataCacheEntry`, etc.

4. Implement the following two cache simulation functions in the code. Comments are given in the code and read carefully. These two functions are very similar to the `FetchInstructionWord` function. After your implementation, you should compile and run your program. If your implementation is correct, you should see the successful message.

```
int ReadDataWord(int addr)
void WriteDataWord(unsigned int addr, unsigned int word)
```

Grade distribution

```
controlAndRegisterFetch: 20
EXE: 10
MEM: 10
WB: 10
ReadDataWord: 40
WriteDataWord: 20 (10 points bonus)
```

Submission: Upload on dropbox your completed cpusim_cachesim.c file and the cpusim_trace.txt trace file generated by running your program with test256.asm.bin as input.