Principle of Computer Organization

**Implementation of a Single Cycle CPU simulator**

**Project due: 30 November, 23:59pm**

1. **Introduction**

In this project, you are going to implement a single cycle CPU simulator called MiniCPU using C language. Your MiniCPU will demonstrate some functions of MIPS processors as well as the principle of the datapath and the control signals. MiniCPU should read in a file containing MIPS machine codes (in the format specified below) and simulate what the MIPS processor does cycle-by-cycle. A C file called component.c will be provided to you which implementing each component of the single-cycle datapath, you are required to modify and fill in the body of the functions in this file.

1. **Specification of the simulator**
   1. **Instructions to be simulated**

The 14 instructions listed in Figure 1 in the appendix. Note that you are NOT required to treat situations leading to exception.

* 1. **Registers to be handled**

MiniCPU should handle the 32 general purpose registers. At the start of the program, the registers are initialized to be the values specified in *spimcore.c*

* 1. **Memory usage**
* The size of memory of MiniCPU is 64kB (Address 0x0000 to 0xFFFF).
* The system assumes that all program starts at memory location 0x4000.
* All instructions are word-aligned in the memory, i.e. the addresses of all instructions are multiple of 4.
* The simulator (and the MIPS processor itself) treats the memory as one segment. (The division of memory into text, data and stack segments is only done by the compiler/assembler.)
* At the start of the program, all memory are initialized to zero, except those specified in the “-asc” file, as shown in the provided codes.
* The memory is in the following format:  
  e.g. Store a 32-bit number 0x*aabbccdd* in memory address 0x0 – 0x3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Mem[0] |  |  |  |
| Address | 0x0 | 0x1 | 0x2 | 0x3 |
| Content | aa | bb | cc | dd |

* 1. **Conditions that the MiniCPU should halt**

If one of the following situations is encountered, the global flag Halt is set to 1, and hence the simulation halts.

* An illegal instruction is encountered. Instructions beyond the list of instructions in Figure 1 are illegal.
* Jumping to an address that is not word-aligned (being multiple of 4)
* The address of lw or sw is not word-aligned
* Accessing data or jump to address that is beyond the memory.
  1. **Format of the input machine code file**

MiniCPU takes hexadecimal formatted machine codes, with filename *xxx.asc*, as input. An example of *.asc* file is shown below. Code after “#” on any line is treated as comments.

20010000 #addi $1, $0, 0  
200200c8 #addi $2, $0, 200  
10220003 #beq $1, $2, 3  
00000020 #delay slot  
20210001 #addi $1, $1, 1  
00000020 #no operation

The simulation ends when an illegal instruction, such as 0x00000000, is encountered.

* 1. **Note on branch addressing**

The branch offset in MIPS, and hence in MiniCPU*,* is relative to the next instruction, i.e. (PC+4). For example,





1. **Resources**
   1. **Files provided**

Please download project.zip from <ftp://ftp.must.edu.mo/>, the following are files after unzip:

**spimcore.c spimcore.h component.c spimasm.pl incommand**

**test01.asm test01.asc test02.asm test02.asc**

These files contain the main program and the other supporting functions of the simulator. The code should be self-explanatory. You are required to fill in and modify the functions in component.c. **You are not allowed to modify spimcore.c and spimcore.h. All your works should be placed in component.c only. You are not allowed to add new files. Otherwise, your program will not be marked.**

* 1. **MIPS assembler**

A simple assembler spimasm.pl is provided for your convenience of testing your MinCPU. The command is:

$spimasm.pl filename.asm > filename.asc

where filename.asm is your assembly code file and filename.asc is the output machine code file in hexadecimal format.

1. **Functions in component.c**

Firstly, you are required to complete a function (ALU(…)) in component.c that simulates the operations of an ALU.

**void ALU(unsigned A, unsigned B, char ALUControl, unsigned \*ALUresult, char \*Zero)**

**{**

**if(ALUControl==0x0)\*ALUresult=A+B; //add**

**}**

* ALU(…)
  + - 1. Implement the operations on input parameters *A* and *B* according to *ALUControl*.
      2. Output the result to *ALUresult*.
      3. Assign *Zero* to 1 if the result is zero; otherwise, assign 0.
      4. The following table shows the operations of the ALU.

|  |  |
| --- | --- |
| ALUControl | Meaning |
| 000 | Z = A + B |
| 001 | Z = A – B |
| 010 | if A < B, Z = 1; otherwise, Z = 0 |
| 011 | if A < B, Z = 1; otherwise, Z = 0 (A and B are unsigned integers) |
| 100 | Z = A AND B |
| 101 | Z = A OR B |
| 110 | Shift B left by 16 bits |
| 111 | Z = NOT A |

Secondly, you are required to fill in 9 functions in component.c. Each function simulates the operations of a section of the datapath. Figure 2 in the appendix shows the datapath and the sections of the datapath you need to simulate.

In spimcore.c, the function Step() is the core function of the MiniCPU. This function invokes the 9 functions that you are required to implement to simulate the signals and data passing between the components of the datapath. Read Step() thoroughly in order to understand the signals and data passing, and implement the 9 functions.

The following shows the specifications of the 9 functions:

* instruction\_fetch(…)

1. Fetch the instruction addressed by *PC* from *Mem* and write it to *instruction*.
2. Return 1 if an invalid instruction is encountered; otherwise, return 0.

**int instruction\_fetch(unsigned PC, unsigned \*Mem, unsigned \*instruction)**

**{  
 \*instruction=Mem[PC>>2];**

**return 0;**

**}**

* instruction\_partition(…)

1. Partition *instruction* into several parts (*op*, *r1*, *r2*, *r3*, *funct*, *offset*, *jsec*).
2. Read line 41 to 47 of spimcore.c for more information.

**void instruction\_partition(unsigned instruction, unsigned \*op, unsigned \*r1, unsigned \*r2, unsigned \*r3, unsigned \*funct, unsigned \*offset, unsigned \*jsec)**

**{**

**\*op = instruction >> 26;**

**}**

* instruction\_decode(…)

1. Decode the instruction based on opcode (*op*).
2. Assign appropriate values to the variables (control signals) in the structure *controls*.

The meanings of the values of the control signals:

For *MemRead*, *MemWrite* or *RegWrite*, the value 1 means that enabled, 0 means that disabled, 2 means “don’t care”.

For *RegDst*, *Jump*, *Branch*, *MemtoReg* or *ALUSrc*, the value 0 or 1 indicates the selected path of the multiplexer; 2 means “don’t care”.

The following table shows the meaning of the values of ALUOp.

|  |  |
| --- | --- |
| value (binary) | Meaning |
| 000 | ALU will do addition or “don’t care” |
| 001 | ALU will do subtraction |
| 010 | ALU will do “set less than” operation |
| 011 | ALU will do “set less than unsigned” operation |
| 100 | ALU will do “and” operation |
| 101 | ALU will do “or” operation |
| 110 | ALU will shift left *extended\_value* by 16 bits |
| 111 | The instruction is an R-type instruction |

1. Return 1 if a halt condition occurs; otherwise, return 0.

**int instruction\_decode(unsigned op, struct\_controls \*controls)**

**{**

**if(op==0x0){ //R-format**

**controls->RegWrite = 1;**

**controls->RegDst = 1;**

**controls->ALUOp = 7;**

**}**

**else return 1; //invalid instruction**

**return 0;**

**}**

* read\_register(…)

1. Read the registers addressed by *r1* and *r2* from *Reg*, and write the read values to *data1* and *data2* respectively.

**void read\_register(unsigned r1, unsigned r2, unsigned \*Reg, unsigned \*data1, unsigned \*data2)**

**{**

**\*data1 = Reg[r1];**

**}**

* sign\_extend(…)

1. Assign the sign-extended value of *offset* to *extended\_value*.

**void sign\_extend(unsigned offset, unsigned \*extended\_value)**

**{**

**}**

* ALU\_operations(…)

1. Based on *ALUOp* and *funct*, perform ALU operations on *data1*, and *data2* or *extended\_value*.
2. Call the function *ALU(…)* to perform the actual ALU operation.
3. Output the result to *ALUresult*.
4. Return 1 if a halt condition occurs; otherwise, return 0.

**int ALU\_operations(unsigned data1, unsigned data2, unsigned extended\_value, unsigned funct, char ALUOp, char ALUSrc, unsigned \*ALUresult, char \*Zero)**

**{**

**switch(ALUOp){**

**// R-type**

**case 7:**

**// funct = 0x20 = 32, add**

**if (funct==0x20) ALU(data1, data2, 0x0, ALUresult, Zero);**

**else return 1; //invalid funct**

**break;**

**default:**

**return 1; //invalid ALUop**

**}**

**return 0;**

**}**

* rw\_memory(…)

1. Base on the value of MemWrite or MemRead to determine memory write operation or memory read operation.
2. Read the content of the memory location addressed by *ALUresult* to *memdata*.
3. Write the value of *data2* to the memory location addressed by *ALUresult*.
4. Return 1 if a halt condition occurs; otherwise, return 0.

**int rw\_memory(unsigned ALUresult, unsigned data2, char MemWrite, char MemRead, unsigned \*memdata, unsigned \*Mem)**

**{**

**if (MemRead==1){**

**\*memdata = Mem[ALUresult>>2];**

**}**

**return 0;**

**}**

* write\_register(…)

1. Write the data (ALUresult or memdata) to a register (*Reg*) addressed by *r2* or *r3*.

**void write\_register(unsigned r2, unsigned r3, unsigned memdata, unsigned ALUresult, char RegWrite, char RegDst, char MemtoReg, unsigned \*Reg)**

**{**

**Reg[r2] = memdata;**

**}**

* PC\_update(…)

1. Update the program counter (PC).

**void PC\_update(unsigned jsec, unsigned extended\_value, char Branch, char Jump, char Zero, unsigned \*PC)**

**{**

**\*PC+=4;**

**}**

The file spimcore.h is the header file which contains the definition of a structure storing the control signals and the prototypes of the above functions. The functions may contain some parameters. Read spimcore.h for more information.

**5. Notes**

1. This project will be compiled and marked using Eclipse. You can download it from the web (<http://www.eclipse.org/downloads/>) and install it on your computer. Remember you should download and install Eclipse for C/C++.
2. Some instructions may try to write to the register $zero and we assume that they are valid. However, your simulator should always keep the value of $zero 0.
3. You should not do any “print” or ”printf()” operation in component.c; otherwise, the operation will disturb the marking process and your marks will be deducted.
4. To run the compiled executable:
   1. In Windows, open a command prompt.
   2. Go to your working directory.
   3. Type: *your\_executable input\_asc\_file < incommand*

Where *incommand* is the downloaded file. The output shows the values of all registers and memory locations, which allows you to check if your simulator can produce the correct result or not.

1. To debug your program, check if the values of all registers and memory match the assembly program. The following is a sample output:



1. To submit your work, at the beginning your component.c, type in your **English** name, e.g.

/\*

\*

\* Designer: name, student id, email address

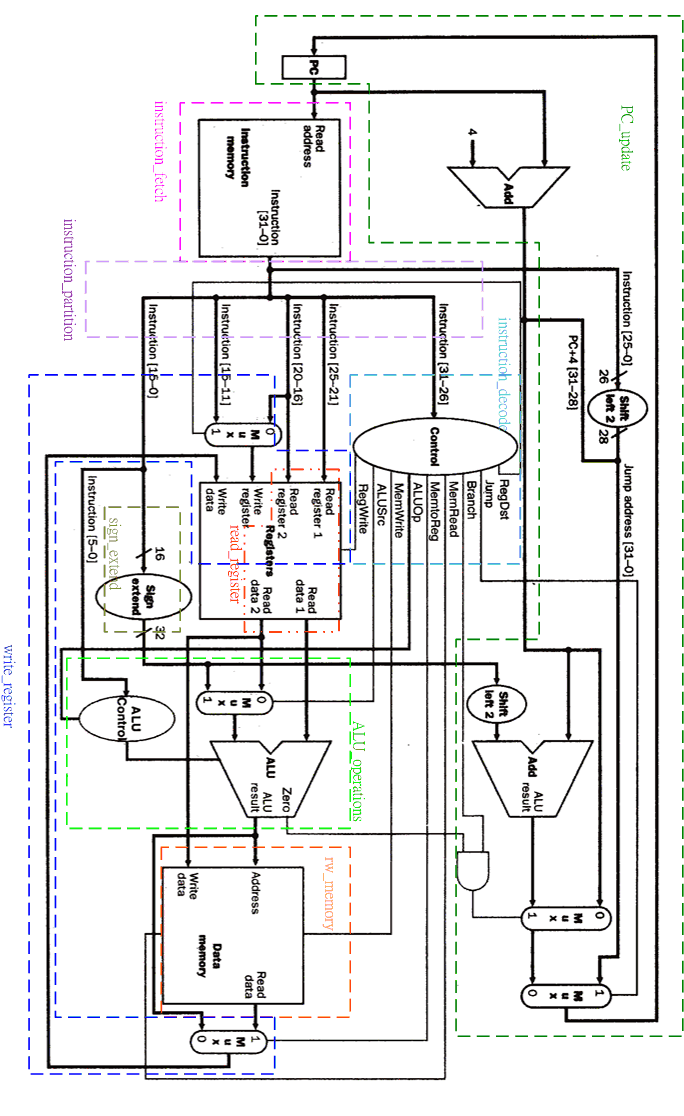
\*/

Email your *component.c* to [yyliang.fit.must@gmail.com](mailto:yyliang.fit.must@gmail.com). The subject of your email must be: **CO001 Project student\_name**.

**Appendix**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Category*** | ***Instruction*** | ***Example*** | ***Meaning*** | ***Comments*** |
| Arithmetic | add | add $s1,$s2,$s3 | $s1 = $s2 + $s3 | 3 operands; overflow detected |
|  | subtract | sub $s1,$s2,$s3 | $s1 = $s2 - $s3 | 3 operands; overflow detected |
|  | add immediate | addi $s1,$s2,100 | $s1 = $s2 + 100 | + constant; overflow detected |
| Logic | and | and $s1,$s2,$s3 | $s1 = $s2 & $s3 | 3 operands; logical AND |
|  | or | or $s1,$s2,$s3 | $s1 = $s2 | $s3 | 3 operands; logical OR |
| Data transfer | load word | lw $s1,100($s2) | $s1 = Memory[$s2 + 100] | word from memory to register |
|  | store word | sw $s1,100($s2) | Memory[$s2 + 100] = $s1 | word from register to memory |
|  | load upper immediate | lui $s1,100 | $s1 = 100 \* 2^16 | loads constant in upper 16 bits |
| Conditional branch | branch on equal | beq $s1,$s2,25 | if ($s1 == $s2)  goto PC + 4 + 100 | equal test; PC relative branch |
|  | set on less than | slt $s1,$s2,$s3 | if ($s2 < $s3) $s1 = 1  else $s1 =0 | compare less than; two's complement |
|  | set less than immediate | slti $s1,$s2,100 | if ($s2 < 100) $s1 = 1  else $s1 =0 | compare < constant; two's complement |
|  | set less than unsigned | sltu $s1,$s2,$s3 | if ($s2 < $s3) $s1 = 1  else $s1 =0 | compare less than; natural number |
|  | set less than immediate unsigned | sltiu $s1,$s2,100 | if ($s2 < 100) $s1 = 1  else $s1 =0 | compare < constant; natural number |
| Unconditional branch | Jump | j label | goto label | Jump to target |

**Figure 1: Instructions to be implemented.**



**Figure 2: The single-cycle datapath to be implemented.**