**Project 1 Design Document**

**Project Specification:**

The files sim.c, computer.h, and computer.c comprise a framework for a MIPS simulator. Complete the program by adding code to computer.c. Your simulator must b3 able to simulate the machine code versions of MIPS instructions. Computer.c needs Decode, Print Instruction, Execute, Update Pc, Memory, and RegWrite to be implemented.

**Decode:**

*Correctness Constraint:*

The “Decode” function should take in arguments unsigned int instr, DecodedInst object d, and RegVals object rVals. The function should first identify the instruction type using the opcode (first 6 bits) and then partition the remaining 26 bits respectively. At termination, the function should have stored strings in rVals and instr that represent the MIPS instruction.

*Structure of Solution:*

To create an encapsulated, and efficient “Decode” function we utilized the bitwise ‘&’ operator to create a “bitmask” which is used in the partitioning portion of the function. The first step is for the op code to be identified. By shifting the instruction 26 bits, we are left with the last 6 bits which represent opcode.

Next, we will have a switch that has cases R, J, and I which will represent the types of instructions. Once the instruction type is identified, we can use “bitmasks” to isolate the relevant bits for decoding. Our bitmasks work by isolating 5 bits, and then shifting them so that they are the left most bits. For rs, we use a bitmap (inst & 0x03e00000) and then shift it by 21 bits. We chose this number because it represents the largest possible value for a register in Rs (which is 31). The following is the conversion to hex from the binary representation of an R-type instruction with Rs = $31:

000000 11111 00000 00000 00000 000000

= 0x03e00000

It is important that we have all the other bytes set to 0, because when we use the bitwise ‘&’ operator, the returned value will only include the two bytes we care about. This is called a “bitmask” because it essentially “masks” the bits we are not interested in. This is particularly useful for “Decode” because it allows us to translate the specific registers. The following is an example of how the “bitmask” would work:

0x12a01002

& 0x03e00000

----------------------------

000100 10101 00000 00010 00000 000010

& 000000 11111 00000 00000 00000 000000

= 000000 10101 00000 00000 00000 000000

= 0x02a00000

Once we have applied the “bitmask”, we simply shift the bytes respectively. For Rs, we would have a shift of 21 bits because we only care about bits 25-21. For Rt we would shift 16, Rd would shift 11, and Shamt would shift 6. Funct would still require a mask, but would not need to be shifted for it is already the rightmost bit.

*Pseudocode:*

Decode ( **unsigned** **int** instr, DecodedInstr\* d, RegVals\* rVals) {

d->op = instr >> 26;

rsMask = (instr & 0x03e00000);

rtMask = (instr & 0x001f0000);

rdMask = (instr & 0x0000f800);

shamtMask = (instr & 0x000007c0);

functmask = (instr & 0x0000003f);

addMask = (instr & 0x03ffffff);

immMask = (instr & 0x0000ffff);

switch (instr) {

case R:

d->type = R;

rVals->R\_rs = mips.registers[d->regs.r.rs = rsMask >> 21];

rVals->R\_rt = mips.registers[d->regs.r.rt = rtMask >> 16];

rVals->R\_rd = mips.registers[d->regs.r.rd = rdMask >> 11];

d->regs.r.shamt = shamtMask >> 6;

d->regs.r.funct = functMask;

Break;

case J:

case jal:

d->type = J;

d->regs.j.target = addMask;

Break;

//I-Type

default:

rVals->R\_rs = mips.registers[d->regs.i.rs = rsMask >> 21];

rVals->R\_rt = mips.registers[d->regs.i.rt = rtMask >> 16];

d->regs.i.addr\_or\_immed = immMask;

}

}

*Testing Strategy:*

* Run with several valid machine code instructions of one specific instruction type.
  + One with all R-type
  + One with all I-type
  + One with all J-Type
* Run with invalid values for registers, i.e. using the $0 register as a rd, rs, and rt in one instruction.
* Run a set of instructions with several instruction types.
  + 3 R-type, 3 I-type, 4 J-type

**Print Instruction:**

*Correctness Constraint:*

“Print Instruction” should take in argument DecodedInstr object d and print the instructions and registers in form of strings. Example output:

Addiu $4, $0, 3

Jal 0x00400010

*Structure of Solution:*

To print the instructions correctly, we will use a variable char i to store the name of the instruction used. Using a switch, we can check the op portion of the d struct and save the name of the instruction in the i variable. After all the stores have been done, we can print a standard phrase with appropriate indentation and variable names

*Pseudocode:*

Char i;

switch(d->op)

For all cases of instruction i = instruction name

Print(specified message for R, I, J type

*Testing Strategy:*

* Run test where stored value for d->op is not a value within range of 0x34
* Run test with valid d->op value

**Execute:**

*Correctness Constraint:*

Function should identify instructions that require the ALU, simulate them, and return the value of the operation

*Structure of Solution:*

With the use of several switch cases, we can identify the types of instructions that are in d->op and only write conditions for those operations that require the ALU. Once those instructions are identified, the value of the operation indicated by the instruction is performed and then returned.

*Pseudocode:*

switch(d->op)

Case 0x09 addiu

Return rVals->R\_rs + d->regs.i.addr\_or\_immed;

Case…

*Testing Strategy:*

* Run 5 simple add and subtract instructions that utilize ALU
* Run a mix-matched batch of instructions that use ALU and don’t use ALU
* Run a set of instructions that require no ALU

**UpdatePC:**

*Correctness Constraint:*

Function must correctly increment program counter by 4 each instruction. The function should also take note of any special cases like “jal” where the program counter is incremented in a different way. If the program counter exits the range of accepted values (0x00400000 -> 0x00401000) then the program should exit.

*Structure of Solution:*

Since most instructions just require an increment of 4, at the beginning of the function, we can increment by four. Additionally, four conditional statements are needed to check for when a beq or bne occurs, a J-type instruction occurs, a jump register occurs, or when pc is out of bounds.

Bne and new can be in the same conditional, because as long as the val returned by execute is true, then they have the same effect on program counter.

In the conditional handling whether or not the program counter has exceeded the bounds, we can also include “mips.pc % 4 != 0” to ensure that the program counter adheres to the architecture of memory even if it is inside of the legal bounds.

*Pseudocode:*

Mips.pc += 4;

if(beq or bne)

If (val = true)

Mips.pc += d->regs.i.addr\_or\_immed <<2;

Else if (J type)

Mips.pc = target address;

Else if (jump register)

Mips.pc = mips.registers[d->regs.r.rs]

if(mips.pc < 0x00400000 or mips.pc > 0x0401000 or mips.pc % 4!= 0

exit

*Testing Strategy:*

* Run no unusual pc instructions
* Run with at least one beq, bne, J type, and jump register instruction
* Jump to an address greater than 0x0401000
* Jump to an address less than 0x04000000
* Jump to address 0x040000003

**Memory:**

*Correctness Constraint:*

* Must account for addresses that are outside memory
* Must be word aligned

*Structure of Solution:*

The solution will first check if the given value is assessing invalid memory addresses or is not word aligned. It will then assign -1 to changedMem and check if the opcode is either a load or save. If it is it will perform the corresponding action.

*Pseudocode:*

if ((val < 0x00401000 || val > 0x00403fff) || val % 4 != 0) {

printf("Memory Access Exception at 0x%.8x: address 0x%.8x", mips.pc, val);

exit(0);

}

\*changedMem = -1;

switch (d->op){

case 0x23: //lw

return mips.memory[(val - 0x00400000) >> 2];

case 0x2b: //sw

mips.memory[((\*changedMem = val) - 0x00400000) >> 2] = mips.registers[d->regs.i.rt];

break;

}

return val;

}

*Testing Strategy:*

* Run with address that is out of bounds
* Run with address that is in bounds

**RegWrite:**

*Correctness Constraint:*

* Account for each type of opcode
* Write cases for each type of opcode
* If J-type assign mips.registers[] to 31

*Structure of Solution:*

See if the opcode is an R, J, or I type and then assign changedReg and mips.registers to the corresponding opcode type. If the opcode is jal then assign to 31. If none then make changedReg = -1.

*Pseudocode:*

\*changedReg = -1;

switch (d->opcode) {

case 0x00:

switch(d->regs.r.funct){

case 0x20://add

case 0x21://add unsigned

case 0x02://shift right logical

case 0x24://and

case 0x25://or

case 0x2a://set less than

case 0x22://subtract

case 0x23://subtract unsigned

\*changedReg = d->regs.r.rd;

mips.registers[d->regs.r.rd] = val;

return;

} return;

case 0x03://jal

\*changedReg = 31;

mips.registers[31] = val;

return;

case 0x09://addiu

case 0x0c://andi

case 0x0d://ori

case 0x0f://lui

case 0x23://lw

\*changedReg = d->regs.i.rt;

mips.registers[d->regs.i.rt] = val;

return;

}

*Testing Strategy:*

Test with each type of opcode instruction then see if the assigned mips.registers is correct.