CSE 4302 / CSE 5302 / ECE 5402: Computer Architecture
Assignment #1: Five Cycles per Instruction Simulator

1 What are you expected to deliver for this assignment?

First, perform a *git pull* on your cse4302 cloned GitHub repository. A new directory, named pa1 will be added. It has the necessary files to complete this programming assignment. You will be modifying simulator.c to complete a five-cycles per instruction implementation of the riscy-uconn ISA. The details on the ISA and what modifications you need to make to the simulator.c are outlined in the next sections of this document. Your implementation must be fully functional for all 11 testcases provided in the unittests directory. You are expected to compile each testcase using the assembler provided to you in assembler directory. For example, for j_test1.asm testcase, you will generate a j_test1.out by running it through the assembler. Then, you may run this testcase using the simulator command: ./simulator j_test1.out.

Once you are ready to submit your simulator.c file for this programming assignment, you are required to email it to the TA and the instructor at mohsin.shan, khan@uconn.edu before the deadline. You will receive an acknowledgement email back from the TA if your submission is accepted. It is your responsibility to ensure a timely submission of your programming assignment. In the subject line, you must state PA1 submission :< your fullname >. You will need to schedule a 10-15 minute time slot with the TA to complete a demo of your assignment to get credit. This process must complete within 2 weeks after the deadline for this assignment. More details on the availability of these time slots will be announced as we approach the assignment’s deadline.

2 Introduction to riscy-uconn Instruction Set Architecture (ISA)

2.1 Memory

The memory is partitioned into instructions and data, and its size is limited to 16,384 addresses. A word (four bytes) of instruction or data is stored in any given address location in the memory, leading to a total memory capacity of 65,538 bytes. The machine only supports word addressable memory.

The instructions reside in the first 2048 locations of the memory, starting from address location 0. A total of 8,192 bytes of 2048 instructions can be stored in memory. Each instruction word is read from right to left. The machine implements a program counter (PC) that initially points at address 0, and addresses the first instruction word (or four bytes). The next instruction’s four bytes are stored at address 1, and so on. Each instruction increments the PC by 4 bytes. However, control flow instructions, such as BNE, BEQ, J, JAL and JR potentially modify the PC to a non-sequential instruction address. The address of the instruction memory is always computed by dividing PC by 4. For example, if the PC is calculated as 32, memory address to look up the corresponding instruction word is calculated using 32/4 or 8.

The data resides on address 2,048 through 16,383. Again, each address contains a word (or four bytes) of data. The machine only supports word addressable load and store instructions. Therefore, the address must resolve to one of the unique memory locations from 2,048 to 16,383. The programmer is responsible for calculating the correct address by using the addressing mode available in the LW and SW instructions.

2.2 Registers

The machine implements the MIPS like ISA using 32 registers, where each register is 32-bits (or one word). These registers are named by the ISA as $zero, $at, $v0-1, $a0-a3, $t0-t9, $s0-s7, $k0-k1, $gp, $sp, $fp, and $ra. The $zero register always contains a value of 0.

2.3 Instructions

The machine adopts a subset of the MIPS like ISA. A 32-bit instruction, i[31:0] is broken into three formats:

R-Type:

i[31:26] (6b OP), i[25:21] (5b RS), i[20:16] (5b RT), i[15:11] (5b RD), i[10:6] (5b shamt), i[5:0] (6b func)

I-Type:

i[31:26] (6b OP), i[25:21] (5b RS), i[20:16] (5b RT), i[15:0] (16b IMM)

J-Type:

i[31:26] (6b OP), i[25:0] (26b ADDR)
The OP and func are 6-bit fields that are used to differentiate between instruction types. The RS, RT and RD are 5-bit specifiers for registers that are used as operands or destination for an instruction. These are also referred as $rs$, $rt$, and $rd$ in this document. The shamt is a 6-bit shift used for shift amount in SRL and SLL shift instructions. IMM is a 16-bit immediate value used to extract an operand value directly from the instruction. Finally, the 26-bit ADDR is target field used to determine the PC address for unconditional jumps.

The various instructions supported by this machine are specified in the mipsInstructionMap.h file. You are expected to modify the simulator.c to support all instruction types identified in this file. More details on the implementation of each instruction are described in the coming sections.

3 Assembler

The assembler is provided to you. However, you will need to compile it using the Makefile in the assembler directory. For instructions about how to use the assembler are available in the README.md file.

3.1 Assembler Labels

The assembler converts instructions to machine code. Moreover, .text and .data direct the assembler to start of instructions and data respectively. For example, each instruction following .text is converted into it 32-bit machine code. Moreover, the .data identifies the start of data from 2048 address. Each subsequent data word corresponds to the next sequential address. For example, the third word after .data will have a memory address of 2050. Moreover, the programmer identifies the initial value of any given data word in memory. 2048: .word 10 puts a value of 10 in memory [2048].

4 Simulator

The pa1 directory has a README.md file for details. However, the simulator.c is provided with a skeleton code. It consists of a main function that performs the simulator initialization(), a process_instructions() function that executes instructions, and some debugging functions that output the registers (rdump()) and memory (mdump()) contents. The initialization() function first initializes the 32 registers in the machine, as well as all memory elements. Next, it starts reading the assembler output file (e.g., test.out) starting with the .text section. Each row in the .out file is read one by one and each instruction word is loaded into memory starting address 0. When the .out file reaches the marker, “11111111111111111111111111111111”, it indicates the end of .text section. This marker is ignored and not loaded into memory. However, the 32-bit data in the next row following this marker is the data memory word to be loaded starting address 2048. The subsequent rows with valid data are loaded in memory addresses with an increment of +1. The inst_dump(), rdump(), rdump_pt(), and mdump() functions are provide to assist with debugging the design. You are not allowed to modify any of the debugging and initialization functions. By default, they are invoked during the process_instructions() function.

The process_instructions() function is also given, and you are not allowed to modify it. It fetches one instruction at a time using the fetch() function, and sends it to decode(), execute(), memory_stage() and write_back_stage() for instruction processing. The program terminates when addi $0, $0, 1 (an attempt to write 1 to $0 register) instruction is fetched and executed. The fetch() function is provided to you, and you are not allowed to change it. It takes PC/4 and uses it as an address to access an instruction from memory to process. This fetched instruction is then forwarded to the decode(), execute(), memory_stage() and write_back_stage() functions that you will implement in this assignment. Each of these four stages use a struct Registers that incorporates necessary dynamic information for each instruction. This information is outlined in simulator.h, and further elaborated below:

- x — op field
- y — funct field
- rs — rs register specifier
- rt — rt register specifier
- rd — rd register specifier
- sa — shamt value
• imm — immediate value
• memory_flag — identify LW or SW
• addr_mem — memory address for LW and SW instruction types
• jmp_out_31 — return address for JAL
• A — register[rs]
• B — register[rt]
• C — temporary register to hold instruction output value

The functional simulator fetches one instruction at a time and executes it in five cycles. This is accomplished by the process instruction() function, which you are not allowed to modify. However, several functions called by this function are not fully implemented to support this MIPS like ISA instruction processing. This assignment requires you to implement the functionality of all instruction types from mipsInstructionMap.h. You will be modifying the decode(), execute(), memory stage() and write back stage() functions to complete this assignment. You submission is required to not modify any of the other functions and the first 14 lines of simulator.c. Below, each instruction’s functionality is outlined to steer your implementation of the these four phases of instruction processing.

4.1 R-TYPE OPERATIONS

4.1.1 ADD
Add: Adds two registers and stores the result in a register.
syntax: add $d, $s, $t
encoding: 0000 00ss ssst tttt dddd d000 0010 0000
operation: $d = $s + $t
decode(): 6-bit OP and func fields must be decoded to identify this operation. Moreover, 5-bit RS, RT, and RD register specifiers must be extracted. The register[RS] and register[RT] are read to read the two operands. The PC is advanced by 4 bytes using the advance_pc(4) function call.
execute(): The two operands are added using the + operator to compute the output value for this operation.
memory stage(): Nothing is done for this instruction.
write back stage(): The register[Rd] is updated with the output value of this operation.

4.1.2 SUB
Subtract: Subtracts two registers and stores the result in a register.
syntax: sub $d, $s, $t
encoding: 0000 00ss ssst tttt dddd d000 0010 0001
operation: $d = $s - $t
Operation is same as ADD, but during execute() phase the − operator is used to compute the output value for this operation.

4.1.3 AND
Bitwise And: Bitwise ands two registers and stores the result in a register.
syntax: and $d, $s, $t
encoding: 0000 00ss ssst tttt dddd d000 0010 0100
operation: $d = $s & $t
Operation is same as ADD, but during execute() phase the & operator is used to compute the output value for this operation.

4.1.4 OR
Bitwise Or: Bitwise logical ors two registers and stores the result in a register.
syntax: or $d, $s, $t
encoding: 0000 00ss ssst tttt dddd d000 0010 0101
operation: $d = $s | $t
Operation is same as ADD, but during execute() phase the | operator is used to compute the output value for this operation.
4.1.5 SLL
Shift Left Logical: Shifts a register value left by the shift amount listed in the instruction, and places the result in a third register. Zeroes are shifted in.

syntax: sl $d, $t, h
encoding: 0000 00ss ssst tttt dddd dh00 hh00 0000
operation: $d = $t << h
Operation is same as ADD, but during execute() phase, the register[RT] is shifted left by h amount (using the << operator) to compute the output value for this operation.

NOTE: The encoding for a NOOP represents the instruction SLL $0, $0, 0, which has no side effects on the register and memory state of the machine.

4.1.6 SRL
Shift Right Logical: Shifts a register value right by the shift amount (shamt) and places the value in the destination register. Zeroes are shifted in.

syntax: srl $d, $t, h
encoding: 0000 00- - - - -t tttt dddd dh00 hh00 0010
operation: $d = $t >> h
Operation is same as ADD, but during execute() phase, the register[RT] is shifted right by h amount (using the >> operator) to compute the output value for this operation.

4.1.7 SLT
Set on Less than (signed): If $s$ is less than $t$, $d$ is set to one. It gets zero otherwise.

syntax: slt $d, $s, $t
encoding: 0000 00ss ssst tttt dddd d000 0010 1010
operation: if $s < t, then $d = 1, else $d = 0
Operation is same as ADD, but during execute() phase, the if-else check is used to compare register[RS] and register[RT] using the < operation.

4.1.8 JR
Jump Register: Jump to the address contained in register $s$

syntax: jr $s
encoding: 0000 00ss ssst tttt d000 0000 0000 0000 1000
operation: PC = register[RS]

decode(): 6-bit OP and func fields must be decoded to identify this operation. Moreover, 5-bit RS register specifier must be extracted. The register[RS] is read to determine the jump address for changing the control flow of instructions. Instead of using the advance_pc() function to increment PC, the PC is set to register[RS]. Note that jr is generally used in combination with jal instruction to return from a procedure call.

execute(): Nothing is done for this instruction.
memory_stage(): Nothing is done for this instruction.
write_back_stage(): Nothing is done for this instruction.

4.2 I-TYPE OPERATIONS

4.2.1 LW
Load Word: A word is loaded into a register from the specified address.

syntax: lw $t, offset($s)
encoding: 1000 11ss ssst tttt iiiii iiiii iiiii
operation: addr_mem = $s + offset
$t = memory[addr_mem]

Note: The 32-bit word is fetched from memory using an aligned memory address. The programmer is responsible for addressing data memory starting address location 2048 and maximum address of 16,383. The address is calculated by sign-extending the 16-bit offset to the register length (32-bits), and then adding the register[RS] contents to the sign-extended offset.
decode(): 6-bit OP field must be decoded to identify this operation. Moreover, 5-bit RS and RT register specifiers must be extracted. The register[RS] is read to determine the base for address calculation. Moreover, the 16-bits offset (or imm) is captured with top 16 bits zeroed out. The memory_flag is set for later stages to use when performing the LW operation. The PC is advanced by 4 bytes using the advance_pc(4) function.

execute(): The memory address is calculated using the + operator in this stage, and stored in addr_mem.

memory_stage(): The memory lookup is done using memory[addr_mem].

write_back_stage(): The value from the memory lookup is stored in register[RT].

4.2.2 SW

Store Word: The contents of $t$ is stored at the specified memory address.

syntax: sw $t$, offset($s$)

encoding: 1010 11ss ssst tttt iii iii iiii iiii iiii iiii

operation:

addr_mem = $s + offset
memory[addr_mem] = $t$

decode(): 6-bit OP field must be decoded to identify this operation. Moreover, 5-bit RS and RT register specifiers must be extracted. The register[RS] is read to determine the base for address calculation. Moreover, the 16-bits offset (or imm) is captured with top 16 bits zeroed out. The memory_flag is set for later stages to use when performing the SW operation. The PC is advanced by 4 bytes using the advance_pc(4) function.

execute(): The memory address is calculated using the + operator in this stage, and stored in addr_mem.

memory_stage(): The 32-bit value of register[RT] is written to memory[addr_mem].

write_back_stage(): Nothing is done for this instruction.

4.2.3 ANDI

Bitwise and Immediate: Bitwise ands a register and an immediate value and stores the result in a register.

syntax: andi $t$, $s$, imm

encoding: 0011 00ss ssst tttt iii iii iiii iiii iiii iiii iiii

operation:

$t = s & imm$

decode(): 6-bit OP field must be decoded to identify this operation. Moreover, 5-bit RS and RT register specifiers must be extracted. The register[RS] is read for one operand. The second operand is calculated by sign-extending the 16-bit imm field to register length (32-bits). The PC is advanced by 4 bytes using the advance_pc(4) function.

execute(): The two operands are anded using the & operator to compute the output value for this operation.

memory_stage(): Nothing is done for this instruction.

write_back_stage(): The register[RT] is updated with the output value of this operation.

4.2.4 ADDI

Add immediate: Adds a register and a sign-extended immediate value and stores the result in a register.

syntax: addi $t$, $s$, imm

encoding: 0010 00ss ssst tttt iii iii iiii iiii iiii iiii iiii

operation:

$t = s + imm$

Operation is same as ANDI, but during execute() phase the + operator is used to compute the output value for this operation.

4.2.5 ORI

Bitwise or immediate: Bitwise ors a register and an immediate value and stores the result in a register.

syntax: ori $t$, $s$, imm

encoding: 0011 01ss ssst tttt iii iii iiii iiii iiii iiii iiii iiii

operation:

$t = s | imm$

Operation is same as ANDI, but during execute() phase the | operator is used to compute the output value for this operation.
4.2.6 SLTI

Set on less than immediate: If $s$ is less than immediate, $t$ is set to one. It gets zero otherwise.

- **Syntax:** `slti $t, $s, imm`
- **Encoding:** `0010 10ss ssst tttt iiii iiii iiii iiii`
- **Operation:** If $s < \text{imm}$, then $t = 1$, else $t = 0$

Operation is same as ANDI, but during `execute()` phase, the if-else check is used to compare register[RS] and sign-extended imm field using the `<` operation.

4.2.7 LUI

Load upper immediate: The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.

- **Syntax:** `lui $t, imm`
- **Encoding:** `0011 11- - - -t tttt iiii iiii iiii iiii`
- **Operation:** $t = \text{imm} \ll 16$

4.2.8 BEQ

Branch on Equal: Branches if the two registers are equal.

- **Syntax:** `beq $s, $t, offset`
- **Encoding:** `0001 00ss ssst tttt iiii iiii iiii iiii`
- **Operation:** If $s == t$, then `pc = pc + 4 + (16-bit signed offset)`; else `pc = advance_pc(4)`

Note: The 16-bit offset in this instruction is calculated by the assembler using the difference between the 32-bit address of instruction following the BEQ (address of PC+4 due to MIPS semantics), and the address of the Label. For example, if a program wants to loop back seven instructions from BEQ, then the offset will be stored as 0xffffffff0 or -32. When BEQ tests positive for $s == t$, the new PC will be calculated as `PC+4-32` or `PC-28`, which will allow the program to loop back seven instructions (refer to `fetch()`, where a PC/4 is used as index for instruction memory). Similarly, if the program wants to loop forward seven instructions then the offset will be stored as 0x18 or 24. When the BEQ tests positive for $s == t$, the new PC will be calculated as `PC+4+24` or `PC+28`, which will allow the program to loop forward seven instructions.

4.2.9 BNE

Branch on not Equal: Branches if the two registers are not equal.

- **Syntax:** `bne $s, $t, offset`
- **Encoding:** `0001 00ss ssst tttt iiii iiii iiii iiii`
- **Operation:** If $s != t$, then `pc = pc + 4 + (16-bit signed offset)`; else `pc = advance_pc(4)`

Operation is same as BEQ, but during `decode()` phase, the check on register[RS] and register[RT] must resolve to be not equal for the branch to resolve taken.

4.3 J-TYPE OPERATIONS

4.3.1 J

Jump: Jumps to the calculated address

- **Syntax:** `j target`
encoding: 0000 10ii iii ii iii ii iii ii i
operation: PC = 26-bit target appended with six upper bits of zeros
Note: The target will never use more than 11 lower bits since the instruction memory has a fixed range of 2048.

decode(): 6-bit OP field must be decoded to identify this operation. The PC is updated using (instruction_fetch & 0x03FFFFFF). Here, the instruction_fetch is the input to the decode() phase from the fetch() phase. The masking operation clears the upper six bits to zeros and captures the remaining target bits.
execute(): Nothing is done for this instruction.
memory_stage(): Nothing is done for this instruction.
write_back_stage(): Nothing is done for this instruction.

4.3.2 JAL
Jump and Link: Jumps to the calculated address, and stores the return address in $31 (aka $ra).
syntax: jal target
encoding: 0000 11ii iii ii iii i iii i i
operation: $31 = PC + 4
PC = 26-bit target appended with six upper bits of zeros
decode(): 6-bit OP field must be decoded to identify this operation. The PC + 4 is first captured in jmp_out 31. The PC is then updated using (instruction_fetch & 0x03FFFFFF). Here, the instruction_fetch is the input to the decode() phase from the fetch() phase. The masking operation clears the upper six bits to zeros and captures the remaining target bits.
execute(): Nothing is done for this instruction.
memory_stage(): Nothing is done for this instruction.
write_back_stage(): The register[31] is updated with jmp_out 31.

5 Debugging
For debugging your design, you have multiple capabilities. You can use the printf statements in the simulator.c file. We have provided a runtime debug flag on line 11. You can add printf statements as you wish, but (1) they must be qualified with the debug flag, and (2) they must not be inserted in any functions outside of decode(), execute(), memory stage() and write_back stage() functions. In addition, we have provided three functions, rdump(), rdump_pt(), and mdump() to facilitate observations into the register and memory implemented by this machine. They are used main() after all instructions are processed. You can add these facilities at any point in the decode(), execute(), memory stage() and write_back stage() functions.

A facility called pipe trace is added to the simulator to support visualization of instruction processing across cycles. The file pipe trace.txt will be created whenever the simulator is executed. The pipe trace flag in simulator.c represents the code available to you for tracing the pipeline. This capability uses two functions, inst_dump() and rdump_pt() that are provided to you. To enable pipe tracing, you must assert the pipe_trace = 1 at line 12.

Finally, you can use the GDB debugger. After compiling the simulator, you can invoke execution of a testcase under GDB. For example, for j_test1.asm testcase, you will generate a j_test1.out by running it through the assembler. Then, you may invoke GDB, using gdb --args ./simulator j_test1.out. gdb is now waiting for the user to type a command. Type run at the (gdb) prompt. You can use the backtrace and x features of GDB to inspect crashes. Many times you want to watch what the program is doing right before it crashes. One way to do this is to step through, one at a time, each statement in the program until you reach the point in execution where you want to make observations. This may be a slow tedious process, so you may want to just run to the particular section of code and stop execution at that point for further examination. For this you will use the breakpoint feature. Basically, a breakpoint is a line in the source code where the debugger breaks execution. For example, you may want to break at decode function at line 96 (in the lab1 simulator.c given to you). For this you can run break 96 and then run. So when the program is run, it will return control to the debugger every time it reaches line 96. This may not be desirable if the function is called many times but only has problems with certain values that are passed. Conditional breakpoints are useful in this case. For example, say you want to tell the debugger to only break at line 96 if instruction_fetch=20512. This can be done by issuing condition 1 instruction_fetch = 20512 command after setting the breakpoint using break 96, and before running the program using run command. Now, the debugger will only break here when the specified condition is true. After setting the breakpoint appropriately, you can now locate
the source of the error. This is accomplished using the \textit{step} command. \texttt{gdb} has the nice feature that when enter is pressed without typing a command, the last command is automatically used. That way you can step through by simply tapping the enter key after the first step has been entered. \texttt{gdb} can be exited by typing \texttt{quit}. For details on GDB, visit \url{https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_toc.html}.