UIC – ECE 366: Computer Organization - Fall 2018

Project 4: MIPS mini SIM

For this project, you will form your own team (of 1 or 2 members) to write a python program, which takes as input a text file i_mem.txt (containing a MIPS machine code in hex) and output some important information of running this MIPS program:

1. (20%) Result of run: end content of `PC, $1 - $7,` and Dynamic Instruction Count
2. (40%) Cycle number details, in the following cases:
    a. a multi-cycle MIPS CPU:
        i. total # of cycles, breakdown of the 3 / 4 / 5 cycle instructions
        ii. instruction by instruction information: assembly instruction, # of cycle, etc
    b. a pipelined MIPS CPU, assuming:
        - branches are resolved at the 2nd ID stage
        - all the forwarding paths supported to solve data hazard

        i. total # of cycles, breakdown on the total # of cycles of delay due to stalls / flushes to resolve control & data hazards including:
            - lw-use,
            - compute-branch compare
            - branch taken flush
        ii. instruction by instruction information: assembly instruction, detected hazard (solved by fwding without delay / stall with delay / flushing etc)
3. (40%) Cache access behavior of `lw` instructions:
    - for each lw instruction, provide: cache access log (memory address to access, which set / block of the cache is accessed, valid bit & tag info, hit or miss, cache update info
    - overall hit rate of the entire run of the program (hit # / (hit # + miss #))

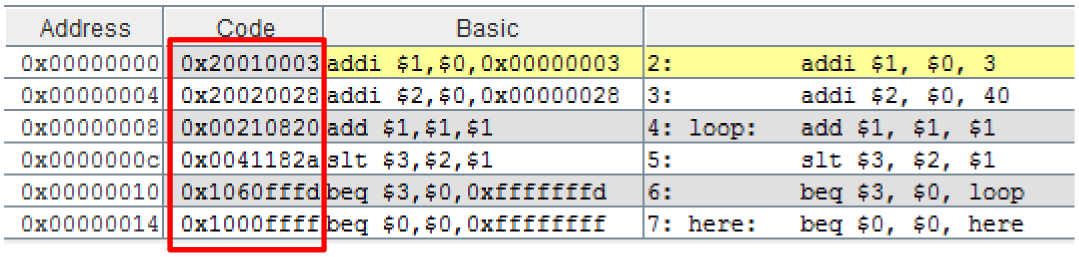
    a. with a direct mapped cache, block size of 4 words, a total of 2 blocks.
    b. with a direct mapped cache, block size of 2 words, a total of 4 blocks.
    c. with a fully-associated cache, block size of 2 words, a total of 4 blocks.
    d. with a 2-way set-associative cache, block size of 2 words, 4 sets, a total of 8 blocks
4. (10%) Extra credit: supporting any set-associative cache configurations by allowing the user to input block size (# of words), # of ways, and # of sets.

Assume the following limited support / subset of MIPS ISA for your program:

- Instructions: `add, sub, xor, addi, beq, bne, slt, lw, sw`
- Registers: `$0 (always = 0), $1 - $7`
- data memory address range: `[0x2000, 0x3000)`
- instruction memory address range: `[0x0000, 0x1000)`
- All the registers / data memory content are initialized to be 0
- The program will end at a dead loop `“label: beq $0, $0, label”` the machine code of which is `0x1000FFFF`

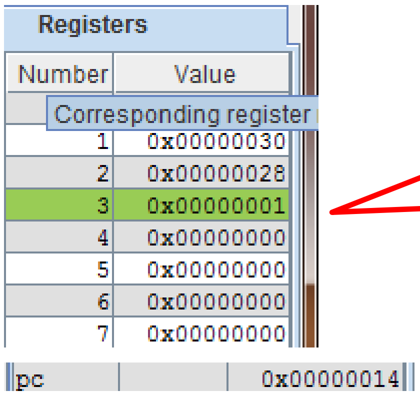
Your python code should be able to read the file containing a valid MIPS program in hex, simulates its running, and correctly output the relevant information.

You should be able to use MARS to partially help verifying your code. For example, the behavior of the following code can be checked by MARs.



Address	Code	Basic		
0x00000000	0x20010003	addi $1,$0,0x00000003	2:	addi $1, $0, 3
0x00000004	0x20020028	addi $2,$0,0x00000028	3:	addi $2, $0, 40
0x00000008	0x00210820	add $1,$1,$1	4: loop:	add $1, $1, $1
0x0000000c	0x0041182a	slt $3,$2,$1	5:	slt $3, $2, $1
0x00000010	0x1060fffd	beq $3,$0,0xfffffffd	6:	beq $3, $0, loop
0x00000014	0x1000ffff	beq $0,$0,0xffffffff	7: here:	beq $0, $0, here

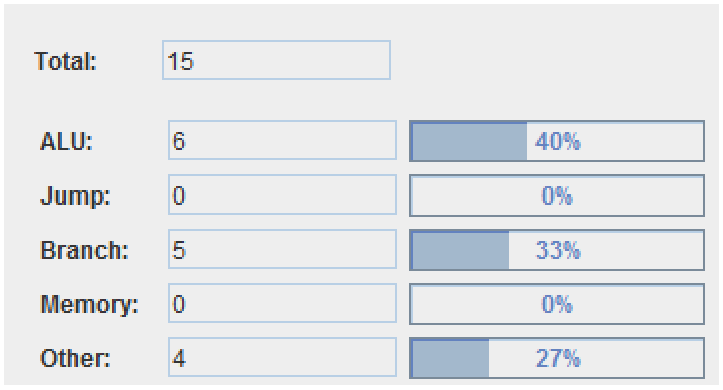
You can use MARS to come up with testing programs and derive machine code. This hex list should be the input file (i_mem.txt) of your Python code.



Registers

Number	Value
	Corresponding register
1	0x00000030
2	0x00000028
3	0x00000001
4	0x00000000
5	0x00000000
6	0x00000000
7	0x00000000
pc	0x00000014

In the end, MARS can tell you the content of $0 - $7, and PC value. Your python code should be able to report the same on screen output.



Total:	15	
ALU:	6	40%
Jump:	0	0%
Branch:	5	33%
Memory:	0	0%
Other:	4	27%

MARS reports the total Dynamic Instruction Count and instruction statistics, but yours should report on screen output, all that is required for single-cycle, multi-cycle, pipelined CPU, and cache behavior.

In addition, you can use **Tools -> Data Cache Simulator** in MARS to check for the cache access behavior of a program.

Submission components:

Include the following files in your Bb submission:

`p4_sim.py:`	Python simulator for your ISA
`p4_output_imem_A1.txt:`	result screen output for program A1
`p4_output_imem_A2.txt:`	result screen output for program A2
`p4_output_imem_B1.txt:`	result screen output for program B1
`p4_output_imem_B2.txt:`	result screen output for program B2

Timeline:

- **[Nov 20th Tu]:** midterm check of each group on GitHub
  - points will be deduced to the groups who have not finished part 1
  - extra credits will be given to the groups who have completed part 2.

- **[Nov 29th Thu]:** final deadline
  - submit all the required files on Bb by 11:59 (end of day), by one of the group members.

Late submission policy and penalty is as below for project 4:

submission by the end of the day of	late penalty on your total score
Fri	5%
Sat	10%
Sun	15%
Mon	20%

Appendix: Program A1, A2, B1, B2

Program A: sw into array, lw and accumulate neg / pos #'s

```
#version 1
addi $1, $0, 2
addi $2, $0, 28

sw_loop:
sw $1, 0x2000($2)
addi $2, $2, -4
beq $2, $0, out
add $1, $1, $1
sub $1, $0, $1
addi $1, $1, 3
beq $3, $3, sw_loop

out:
addi $5, $0, 32

lw_loop:
lw $1, 0x2000($2)
slt $3, $1, $0
beq $3, $0, skip
add $4, $4, $1
skip:
addi $2, $2, 4
bne $2, $5, lw_loop

sw $4, 0x2000($0)
end: beq $0, $0, end
```

```
0x20010002
0x2002001c
0xac412000
0x2042fffc
0x10400004
0x00210820
0x00010822
0x20210003
0x1063fff9
0x20050020
0x8c412000
0x0020182a
0x10600001
0x00812020
0x20420004
0x1445fffa
0xac042000
0x1000ffff
```

```
#version 2
addi $1, $0, 3
addi $2, $0, 80

sw_loop:
sw $1, 0x2000($2)
addi $2, $2, -4
beq $2, $0, out
add $1, $1, $1
sub $1, $0, $1
addi $1, $1, 3
beq $3, $3, sw_loop

out:
addi $5, $0, 40

lw_loop:
lw $1, 0x2000($2)
slt $3, $1, $0
bne $3, $0, skip
add $4, $4, $1
skip:
addi $2, $2, 4
bne $2, $5, lw_loop

sw $4, 0x2000($0)
end: beq $0, $0, end
```

```
0x20010003
0x20020050
0xac412000
0x2042fffc
0x10400004
0x00210820
0x00010822
0x20210003
0x1063fff9
0x20050028
0x8c412000
0x0020182a
0x14600001
0x00812020
0x20420004
0x1445fffa
0xac042000
0x1000ffff
```

Program B: sw into array, iterate lw / sw among neighborhood of 3

```
#version 1
addi $1, $0, 2
addi $2, $0, 28

sw_loop:
sw $1, 0x2000($2)
addi $2, $2, -4
beq $2, $0, out
add $1, $1, $1
sub $1, $0, $1
addi $1, $1, 3
beq $3, $3, sw_loop

out:
addi $6, $0, 40

loop:
add $4, $0, $0
lw $1, 0x2004($2)
add $4, $4, $1
lw $1, 0x2008($2)
add $4, $4, $1
lw $1, 0x200c($2)
add $4, $4, $1
slt $1, $5, $4
beq $1, $0, skip
add $5, $4, $0
skip:
addi $2, $2, 4
bne $2, $6, loop

sw $5, 0x2000($0)
end: beq $0, $0, end
```

```
#version 2
addi $1, $0, 5
addi $2, $0, 60

sw_loop:
sw $1, 0x2000($2)
addi $2, $2, -4
beq $2, $0, out
add $1, $1, $1
sub $1, $0, $1
addi $1, $1, 3
beq $3, $3, sw_loop

out:
addi $6, $0, 60

loop:
add $4, $0, $0
lw $1, 0x2004($2)
xor $4, $4, $1
lw $1, 0x2008($2)
xor $4, $4, $1
lw $1, 0x200c($2)
xor $4, $4, $1
sw $4, 0x2004($2)
xor $5, $5, $4


addi $2, $2, 4
bne $2, $6, loop

sw $5, 0x2000($0)
end: beq $0, $0, end
```