University of Maryland Baltimore County
Department of Computer Science and Electrical Engineering

CMSC 411, Computer Architecture

Term Project Due: December 9, 2017

Objective:

To experience the design issues of advanced computer architectures through the design of a simulator for a simplified MIPS computer using high level programming languages.

Project Statement:

Consider a simplified MIPS computer that follows the design discussed in class and in the textbook while accepting only the following subset of the instructions:

Instruction Class	Instruction Mnemonic
Data Transfers	LW, SW
Arithmetic/ logical	ADD, ADDI, MULT, MULTI, SUB, SUBI, AND, ANDI, OR, ORI, LI, LUI
Control	BEQ, BNE, J
Special purpose	HLT (to halt the simulation)

You need to develop an architecture simulator for the MIPS computer whose organization is shown in Figure 1. The simulator is to accept a program as an input in the MIPS assembly using the above subset of instructions. The simulator should output a file containing the cycle time at which each instruction completes the various stages, and statistics for cache access. The detailed specifications of the input and output files will be provided later in this document. The CPU and Memory system are explained next.

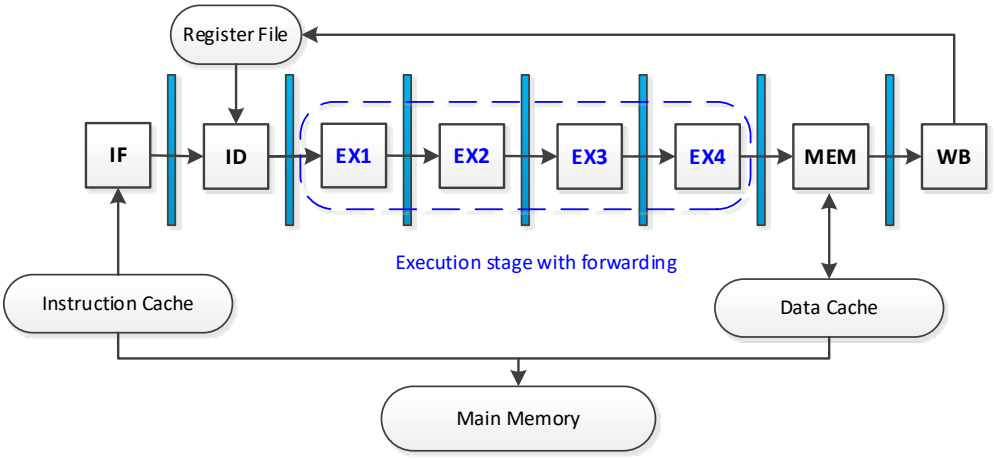

Figure 1: Block diagram description of the organization of considered pipeline processor

Memory: The MIPS machine is assumed to have an instruction cache (I-Cache) with an access time of one cycle. The organization of I-Cache is direct-mapped with four blocks and the block size is four words. In addition, the architecture has a data cache (D-Cache) with hit time of one cycle. D-Cache is a 2-way set associative with a total of four 4-words blocks. A write-allocate placement and a least recently used block replacement strategies are to be applied for D-Cache. A write-back strategy is employed. I-Cache is used in the instruction fetch stage while D-Cache is accessed in the memory stage. Both I-Cache and D-Cache are connected to main memory using a shared bus. In the case of a cache miss, if main memory is busy serving the other cache, we have to wait for it to be free and then start accessing it.

Simultaneous cache misses will be resolved in favor of the I-Cache. In other words, latency of the main memory will be dynamic depending on the time of request and the state of previous requests. Accessing a word in main memory takes 3 cycles

CPU: A multi-stage pipeline processor is employed. Each of the ID and WB pipeline stages complete in one cycle. The time for IF and MEM stages depends on the access time of the I-Cache and D-Cache, respectively. The execution may take up to 4 cycles depending on the instruction. The table below shows the number of cycles each instruction requires in the EX stage before data becomes available:

Number of Cycles	Instructions
0 Cycle	J, BEQ, BNE, LI, LUI
1 Cycle	AND, ANDI, OR, ORI, LW, SW
2 Cycles	ADD, ADDI, SUB, SUBI
4 Cycles	MULT, MULTI

Data forwarding is possible from the MEM, EX2, and EX3 stages to EX1 and from the MEM, EX1, EX2, EX3 stages to the ID stage, and MEM to MEM. For the LI and LUI instructions, data is available for forwarding after the ID stage. Branches are resolved in the ID stage. Meanwhile the IF stage will go ahead and fetch the next instruction, in other words, always "not-taken prediction" will be used in IF stage. If we find out in the ID stage that the outcome of branch is taken, the control unit will flush the IF stage (inserting a bubble) and update the program counter so IF in next cycle will start fetching from the branch target address. When the HLT instruction reaches the ID stage it causes the simulator to terminate any I-cache misses and to flush the IF stage.

You need to take into account the following additional features:

1) Instructions and data are stored in memory starting at address 0x0 and 0x100, respectively.
2) Load and store instructions use word-aligned addresses when accessing data.
3) Both conditional and unconditional jump instructions can be forward and backward. You can assume that a program will not create a closed loop.
4) An instruction stalled for data hazard in the ID stage can get the values in the same cycle WB takes place.
5) The HLT instruction will mark the end of the program, i.e., fetching will seize as soon as the HLT instruction is decoded. In your implementation you can assume that the program will have two HLT instructions at the end in order to stop accessing the cache once the first HLT reaches the ID stage, i.e., the second HLT instruction will be terminated at that time.

The project executable should be named "simulator". The format of the command line shall be as follows:

"simulator inst.txt data.txt output.txt"

- The "*inst.txt*" input is the instruction file and consists of assembly language code, represented as a sequence of instructions in symbolic format such as *LW* or *ADD*. The instructions should be loaded into memory beginning at address 0x00. Your simulator should ignore multiple white-spaces and use "," as the separator for operands. Moreover, there may be *LABELS* before some certain instruction so that branch instructions can easily specify the destination. Every LABEL will be followed by ":" as delimiter.
- The "data.txt" file contains a variable number of 32-bit data words, one per line. These data words are to be placed in memory beginning at memory location 0x100. You can assume that the size of the data segment is 32 words; meaning that the test cases will not require access to more than 32 words of memory. Registers are to be initialized using the LI instruction (in the assembly program in the "*inst.txt*" file).
- The last file is for storing the output of the simulator.

Note: input files' role will be defined based on their position in argument list. The names and their paths might be different and your simulator should not be restricted to specific name(s) or path(s).

The simulator is to be developed in the programming language of your choice. However, you MUST submit a "MAKEFILE" that automates the compilation of your project on the *GL machine*. For those using Java or Python, preparing a "MAKEFILE" could be burdensome. In that case, you MUST submit a simple shell script file named "make.sh" to automate the compilation. Please also include execution syntax in README file, e.g., "java simulator inputFile.asm data.txt output.txt". If you use *ant* and have a *build.xml* file, please make sure to include it in your project.

Input file "inst.txt" format and considerations:

- You will have one instruction on each line.
- We are not going to test your input parser by feeding it with bad input files. However, your program should point out which line of input file is inconsistent with the expected format, and generate an error message and terminate the execution.
- Number of White Spaces (space, tab, enter) should not be a problem for your input parser.
- Your program should not be case sensitive. (e.g ADDI, addi, AddI, Addi, aDdi … are all same)
- You should strongly stick to the format of MIPS instructions. For example if you implement the load immediate as, "LI 1, R1", it will be wrong (the correct format is LI R1, 1).

Output file format:

The simulator must output all desired information to *ONE* output file "*output.txt*". Be sure to follow the output format *EXACTLY*; deviations from this format will hamper the grading process of your submission. The output should contain the following information:

- The instruction and the clock cycle in which it leaves every stage. If an instruction does not enter a particular stage, leave the entry empty. For example, the "BNE" instruction finishes in ID stage and thus there will be no entries in the following stages for this instruction.
- The total number of cache access requests for both instruction and data caches.
- The number of cache hits for both instruction and data caches.

Use the example in project description as your guideline for output file. Do not put extra information such as your name in your output file.

Example:

Consider the following input assembly program (*inst.txt*):

```
        LI      R1, 100h        # addr = 0x100;
        LW      R3, 0(R1)       # boundary = *addr;
        LI      R5, 1           # i = 1;
        LI      R7, 0h          # sum = 0;
        LI      R6, 1h          # factorial = 0x01;
LOOP:   MULT    R6, R5, R6      # factorial *= I;
        ADD     R7, R7, R6      # sum += factorial;
        ADDI    R5, R5, 1h      # i++;
        BNE     R5, R3, LOOP
        HLT
        HLT
```

data.txt

```
00000000000000000000000000000010
00000000000000000000000000000001
00000000000000000000000000000011
00000000000000000000000000010101
00000000000000001111111111111000
00000000000000000000000000000011
00000000000000000000000000000001
00000000000000001111100110100110
00000000000000000001010101010101
00000000000001101101010011110101
00000000000001000010101110010101
00000000000001101101111001010101
00000000000000000101010101110100
00000000000010010101010111110101
00000000001111011110000000000000
00000000000001101101010011110100
00000000000000001111111111111111
00000000000000001111100010100110
00000000000000001010101101110110
00000000001010110100111101011
00000000001000000000000000000111
00000000000000000010000101010101
00000000010011101010100010100
00000001010101010101010101111
00001000100100101011100000001010
00000000000000001111111111111111
00000000000000000010101110101101
00000000000000000000000000000000
00000000000001101101010011110101
00000000000000000000000001100001
00000000000000011000010101010101
00000000000000000000010101010101
```

The ***<u>output</u>*** should be: (Numbers represent the clock cycle that instruction leaves each stage)
Note that the execution cycle number is the cycle of the last stage (=EX4) for all instructions. Branching instructions terminate in the ID stage.

Cycle Number for Each Stage			IF	ID	EX4	MEM	WB
	LI	R1, 100h	12	13	17	18	19
	LW	R3, 0(R1)	13	14	18	39	40
	LI	R5, 1	14	15	39	40	41
	LI	R7, 0h	15	16	40	41	42
	LI	R6, 1h	27	28	41	42	43
LOOP:	MULT	R6, R5, R6	28	29	42	43	44
	ADD	R7, R7, R6	29	42	46	47	48
	ADDI	R5, R5, 1h	42	43	47	48	49
	BNE	R5, R3, LOOP	54	55			
	HLT		55	56			
	HLT		56				

Total number of access requests for instruction cache: 11
Number of instruction cache hits: 8

Total number of access requests for data cache: 1
Number of data cache hits: 0

Execution Trace (to explain the output):

The following is a detailed trace of execution of the above program:

Instructions	Cycles									
	1	2	3	4	5	6	7	8	9	10
LI R1, 100h	stall	stall	stall	Stall	stall	stall	stall	stall	stall	stall
LW R3, 0(R1)										
LI R5, 1										
LI R7, 0h										
LI R6, 1h										

The stall at cycle 1 is caused by an I-Cache miss.

	11	12	13	14	15	16	17	18	19	20
LI R1, 100h	stall	IF	ID	EX1	EX2	EX3	EX4	MEM	WB	
LW R3, 0(R1)			IF	ID	EX1	EX2	EX3	EX4	*stall*	*stall*
LI R5, 1				IF	ID	EX1	EX2	EX3	EX4	
LI R7, 0h					IF	ID	EX1	EX2	EX3	
LI R6, 1h						stall	stall	stall	stall	stall
LOOP: MULT R6, R5, R6										
ADD R7, R7, R6										
ADDI R5, R5, 1h										
BNE R5, R3, LOOP										

The stall caused by the "LW" instruction at cycle 19 blocks the progress of all subsequent instruction in the pipeline.

	21	22	23	24	25	26	27	28	29	30
LI R1, 100h										
LW R3, 0(R1)	*stall*	*stall*	*stall*	*stall*	*stall*	*stall*	*stall*	stall	stall	stall
LI R5, 1										
LI R7, 0h										
LI R6, 1h	stall	stall	stall	stall	stall	stall	IF	ID	EX1	EX2
LOOP: MULT R6, R5, R6								IF	ID	EX1
ADD R7, R7, R6									IF	ID
ADDI R5, R5, 1h										IF
BNE R5, R3, LOOP										
HLT										

		31	32	33	34	35	36	37	38	39	40
LI	R1, 100h										
LW	R3, 0(R1)	stall	stall	stall	stall	stall	stall	stall	stall	MEM	WB
LI	R5, 1										MEM
LI	R7, 0h										EX4
LI	R6, 1h										EX3
MULT	R6, R5, R6										EX2
ADD	R7, R7, R6										stall
ADDI	R5, R5, 1h										stall
BNE	R5, R3,LOOP										stall

		41	42	43	44	45	46	47	48	49	50
LI	R1, 100h										
LW	R3, 0(R1)										
LI	R5, 1	WB									
LI	R7, 0h	MEM	WB								
LI	R6, 1h	EX4	MEM	WB							
MULT	R6, R5, R6	EX3	EX4	MEM	WB						
ADD	R7, R7, R6	stall	stall	EX1	EX2	EX3	EX4	MEM	WB		
ADDI	R5, R5, 1h	stall	stall	ID	Ex1	EX2	EX3	EX4	MEM	WB	
BNE	R5, R3,LOOP			stall	stall	stall	stall	stall	stall	stall	stall
HLT											

		51	52	53	54	55	56						
LI	R1, 100h												
LW	R3, 0(R1)												
LI	R5, 1												
LI	R7, 0h												
LI	R6, 1h												
MULT	R6, R5, R6												
ADD	R7, R7, R6												
ADDI	R5, R5, 1h												
BNE	R5, R3, LOOP	stall	stall	stall	IF	ID							
HLT						IF	ID						
HLT							IF						

There is data dependency and thus the ADD has stall until the value of R6 is calculated by the preceding MULT instruction. The BNE instruction does not cause a jump and the program terminates.

Submission Procedure

You can decide on the number of files you would like to submit for this project. However, please make sure that you also provide a MAKEFILE for the TA to compile and test your code. For instance, suppose you have just one source code file named as ***project.c***, then please write your MAKEFILE as the following format, and make sure you provide the *MAKE CLEAN* function:

```
# CMSC 411, Fall 2017, Term project Makefile

simulator:
	gcc project.c -o simulator

clean:
	-rm simulator *.o core*
```

First you need to ensure the MAKEFILE and the source code files are in the same directory. Then run **make**. An executable named ***simulator*** should appear in the same directory. Ensure that ***simulator*** runs correctly, and then run **make clean**. Check to ensure that ***simulator*** was deleted from the directory.

In order to submit your project:

1- Make sure your project compiles and runs without any problem.
2- Run "make clean" in your project directory to get rid of all the temp and executable files.
3- Rename your project folder to your UMBC user name, e.g. cpailom1.
4- Make sure there are no extra files/directories inside your project folder.
5- ZIP the folder with any program that you have access to. "rar", "tar" and other extensions are not accepted. ONLY standard zip. (now you will have cpailom1.zip)
6- Submit your zip file (cpailom1.zip) via blackboard. Make sure your upload has worked fine by double checking that you can download and manipulate your submitted zip file (this makes sure there is no data corruption).

Please DO NOT email your project submissions to the TA or the instructor.

Instruction Format and Semantics:

Example Instruction	Instruction Name	Meaning
LW R1, 30(R2)	Load word	Regs[R1]<- Mem[30+Regs[R2]]
SW R3, 500(R4)	Store word	Mem[500+regs[R4]]<-Regs[R3]
LI R8, 42	Load immediate	Regs[R8] <- 42
LI R8, -42	Load immediate *	Regs[R8] <- (-42)
LUI R5, 2	Load immediate	Regs[R5] <- 512 (2 * 256)
ADD R1,R2,R3	Add signed	Regs[R1] <- Regs[R2]+ Regs[R3]
ADDI R1,R2, 3	Add immediate signed	Regs[R1] <- Regs[R2]+3
SUB R1,R2,R3	Sub signed	Regs[R1] <- Regs[R2] - Regs[R3]
SUBI R1,R2, 3	Sub immediate signed	Regs[R1] <- Regs[R2] - 3
AND R1,R2,R3	Bitwise AND	Regs[R1] <- Regs[R2] &Regs[R3]
ANDI R1,R2, 3	Bitwise AND-immediate	Regs[R1] <- Regs[R2] & 3
OR R1,R2,R3	Bitwise OR	Regs[R1] <- Regs[R2] |Regs[R3]
ORI R1,R2, 3	Bitwise OR-immediate	Regs[R1] <- Regs[R2] | 3
J LABEL	Unconditional jump	PC <-LABEL
BNE R3, R4, name	Branch not equal	If(R3 != R4) PC<-name
BEQ R3, R4, name	Branch equal	If(R3 == R4) PC<-name
MULT R1, R2, R3	Multiply signed	Regs[R1] <- Regs[R2] * Regs[R3]
MULTI R1, R2, 3	Multiply immediate signed	Regs[R1] <- Regs[R2] * 3

** Immediate values can be positive or negative*