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

CMSC 411, Computer Architecture

Term Project Due: December 6, 2021

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, LI
Arithmetic/ logical	ADD, ADDI, MULT, MULTI, SUB, SUBI
Control	BEQ, BNE, J
Special purpose	HLT (to halt the simulation)

Develop an architecture simulator for the MIPS computer that accepts as an input a program in the MIPS assembly using the above subset of instructions. The output of simulator will be a file containing the cycle time of each instruction in every stage. It also has to report the instruction and data cache accesses as output example that comes later.

The cycle time is 20 ns allowing the ID and WB pipeline stages to be completed in ***one cycle***. The execution stage may take 13 cycles depending on the instruction. The MIPS machine is assumed to have an instruction cache (I-Cache) with an access time of 20 ns (one cycle). The organization of I-Cache is direct-mapped with 4 blocks and the block size is 4 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 least recently used block replacement strategy is to be applied for the D-Cache. The I-Cache and D-Cache are both connected to main memory using a shared bus. In the case of a cache miss, if main memory is busy with the other cache we have to wait for it to be free and then start accessing it. In other words, latency of the main memory will be dynamic depending on the time of request and the state of previous requests. The main memory is accessible through one-word-wide bus, and its access time is 60ns (3 cycles). A write-through strategy is pursued for the D-Cache with No-write allocate. Assume that the write buffer is large enough so that no pipeline stalls will be experienced due to buffer overflow. The words in the buffer will be

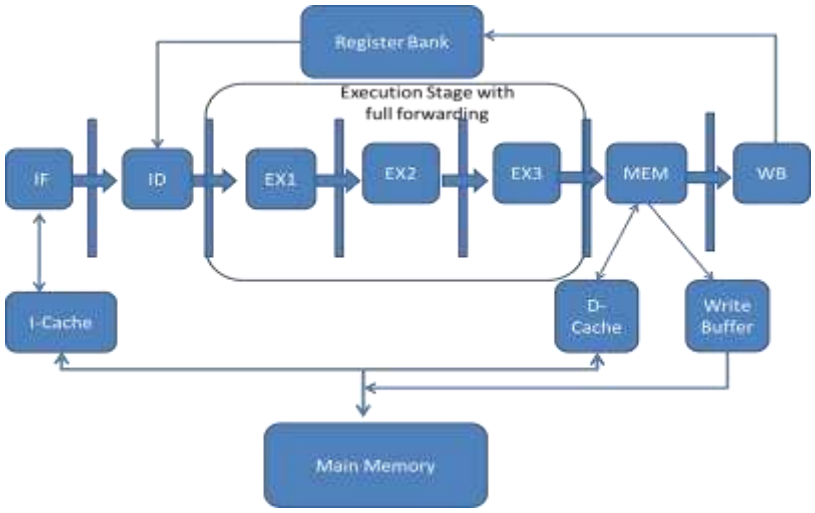

Figure 1: Block diagram description of the organization of the pipeline processor

flushed (written to memory) in a FIFO order while memory is not in use. Cache misses will be given priority over flushing the write-buffer (unless during the 3-cycles need for transferring one word from the buffer to memory). It is worth noting that writing to a word which exists in the buffer will just update the contents and will not force flushing the buffer. Read access for unwritten entries in the buffer will force a flush of the buffer until that entry, and trigger a cache miss to bring its block from memory. Writing to a block, which was brought in based on a prior read access, requires the block to be invalidated and the data to be written to the buffer. The table below shows the number of cycles each instruction takes in the EX stage (in case data needs to be forwarded).

Number of Cycles	Instructions
1 Cycle	BEQ, BNE, J, LW, SW, LI
2 Cycles	ADD, ADDI, SUB, SUBI
3 Cycles	MULT, MULTI

Data forwarding is possible from the MEM stage to the EX 1 & ID stages, from the EX 1, 2, 3 stages to ID stage, from EX 2, 3 to EX 1. 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 a 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.

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

"simulator inst.txt data_segment.txt output.txt"

- The first input file should be the instruction file "*inst.txt*" that consists of assembly language code beginning at memory location 0x0 and is based on Table 1. Your program should ignore multiple white-spaces and use "," as the separator for operands. Moreover, there should be *LABELS* before some certain instruction so that branch instructions can easily specify the destination. The delimiter for separating Labels, operation, and operands is a "|".
- The second input file should contain data words to be placed in memory beginning at memory location 0x100. You can assume that the size of data segment is 32 words; meaning that the test cases will not require access to more than 32 words of memory. Registers will be initialized using the LI instruction.
- The last file is for storing the output of the simulator in.

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. For those using Java or Python, not C, making a "MAKEFILE" could be quite 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. For example, "java simulator inputFile.asm data.txt output.txt". If you used ant and have a build.xml file, please make sure to include it in your project.

Input files 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.
- 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:

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:

```
        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
```

data_segment.txt

```
00000000000000000000000000000010
00000000000000000000000000000001
00000000000000000000000000000011
00000000000000000000000000010101
00000000000000001111111111111000
00000000000000000000000000000011
00000000000000000000000000000001
00000000000000001111100110100110
00000000000000000001010101010101
00000000000001101101010011110101
00000000000001000010101110010101
00000000000001101101111001010101
00000000000000000101010101110100
00000000000001001010101011110101
00000000001111101110000000000000
00000000000001101101010011110100
00000000000000001111111111111111
00000000000000001111100010100110
00000000000000001010101101110110
00000000001010110100111110101011
00000000001000000000000000000111
00000000000000000010000101010101
00000000010011101010101000010100
00000001010101010101010101011111
00001000100100101011100000001010
00000000000000001111111111111111
00000000000000000010101110101101
00000000000000000000000000000000
00000000000001101101010011110101
00000000000000000000000000100001
00000000000000011000010101010101
```

00000000000000000000010101010101

The *output* 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 (=EX3) for all instructions. Branching instructions terminate in the EX1 stage and does not have entries in the MEM and WB stages.

Cycle Number for Each Stage			IF	ID	EX3	MEM	WB
	LI	R1, 100h	13	14	17	18	19
	LW	R3, 0(R1)	14	15	18	41	42
	LI	R5, 1	15	16	41	42	43
	LI	R7, 0h	16	17	42	43	44
	LI	R6, 1h	29	30	43	44	45
LOOP:	MULT	R6, R5, R6	30	41	44	45	46
	ADD	R7, R7, R6	41	42	47	48	49
	ADDI	R5, R5, 1h	42	45	48	49	50
	BNE	R5, R3, LOOP	55	56	59		
	HLT		56	59			

Total number of access requests for instruction cache: 10
Number of instruction cache hits: 7

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	stall	IF	ID	EX1	EX2	EX3	MEM	WB	
LW	R3, 0(R1)				IF	ID	EX1	EX2	EX3	*stall*	*stall*

	LI	R5, 1					IF	ID	EX1	EX2		
	LI	R7, 0h						IF	ID	EX1		
	LI	R6, 1h							stall	stall	stall	stall
LOOP:	MULT	R6, R5, R6										
	ADD	R7, R7, R6										
	ADDI	R5, R5, 1h										
	BNE	R5, R3, LOOP										

Note that at cycle 11 stalls occur due to an I-cache miss. While doing that, stalls due to D-cache cannot be handled at cycle 14 due to bus contention while the I-cache miss is handled. At cycle 17, the D-Cache miss is handled. The stall caused by the "LW" instruction at cycle 14 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	stall	stall	IF	ID
LOOP:	MULT	R6, R5, R6										IF
	ADD	R7, R7, R6										
	ADDI	R5, R5, 1h										
	BNE	R5, R3, LOOP										
	HLT											

Note that data dependency so far did not cause stalls and resolved by data forwarding.

		31	32-40	41	42	43	44	45	46	47	48
LI	R1, 100h										
LW	R3, 0(R1)	stall	stall	MEM	WB						
LI	R5, 1			EX3	MEM	WB					
LI	R7, 0h			EX2	EX3	MEM	WB				
LI	R6, 1h			EX1	EX2	EX3	MEM	WB			
MULT	R6, R5, R6			ID	EX1	EX2	EX3	MEM	WB		

ADD	R7, R7, R6			IF	ID	stall	stall	EX1	EX2	EX3	MEM
ADDI	R5, R5, 1h				IF	stall	stall	ID	Ex1	EX2	EX3
BNE	R5, R3,LOOP							stall	stall	stall	stall
HLT											

		49	50	51	52	53	54	55	56	57	58	59	60
LI	R1, 100h												
LW	R3, 0(R1)												
LI	R5, 1												
LI	R7, 0h												
LI	R6, 1h												
MULT	R6, R5, R6												
ADD	R7, R7, R6	WB											
ADDI	R5, R5, 1h	MEM	WB										
BNE	R5, R3, LOOP	stall	stall	stall	stall	stall	stall	stall	stall	IF	ID	EX1	
HLT											IF	ID	

There is data dependency and thus the ADD has stall until the value of R6 is calculated by the preceding MULT instruction. The BNE causes stalls at the IF stage due to an I-Cache miss. Notice stalls in red (and italic) means that it is waiting for something else to finish. 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, Spring 2011, 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.

Submit ***all of your project files*** in a zip file to blackboard:

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)
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, constant	Bitwise OR-immediate	Regs[R1] <- Regs[R2] | constant
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, constant	Multiply immediate signed	Regs[R1] <- Regs[R2] * constant

** Immediate values can be positive or negative*