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

CMSC 611 – Advanced Computer Architecture

Class Project Due: *December 3rd, 2019*

Objective

To experience the design issues of advanced computer architectures through the design of an analyzer for a simplified MIPS CPU using high level programming languages. The considered MIPS CPU adopts a multi-cycle pipeline processor to dynamically schedule instruction execution and employs caches in order to expedite memory access.

Project Statement

Consider a simplified version of the MIPS instruction set architecture shown below in Table 1 and whose formats are provided at the end of this document.

Table 1: Reduced MIPS instruction set

Instruction Class	Instruction Mnemonic
Data Transfers	LW, SW, L.D, S.D
Arithmetic/ logical	DADD, DADDI, DSUB, DSUBI, AND, ANDI, OR, ORI, ADD.D, MUL.D, DIV.D, SUB.D
Control	J, BEQ, BNE
Special purpose	HLT (to stop fetching new instructions)

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 subset of instructions in Table 1. The output of simulator will be 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 following explains the CPU and Memory system:

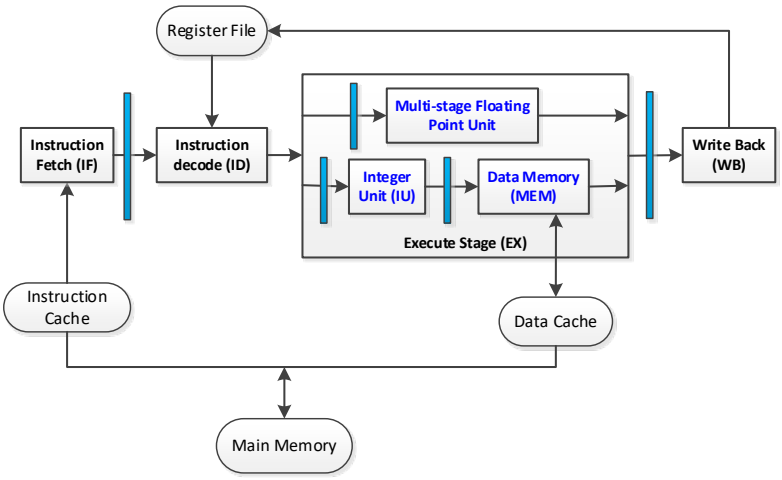

Figure 1: Block diagram description of a MIPS computer

Memory: The MIPS machine has an instruction cache (I-Cache) organized as direct-mapped with 4 blocks and the block size is 4 words. In addition, the machine has a data cache (D-Cache). 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 D-Cache. A write-back strategy is employed with a write-allocate policy. 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. In other words, latency of the main memory will be dynamic depending on the time of a request and the state of previous requests. If both caches experience a miss at the same cycle, the I-Cache will be given priority. The main memory is accessible through one-word-wide bus. The access time for memory, I-Cache (hit

time) and D-Cache (hit time) are specified as input in a configuration file, named "config.txt". Memory is 2-way interleaved making the access time for 2 consecutive words equals T+k cycles, where T and k are the access time of memory and cache, respectively. All access to memory will be word-aligned. Thus, the cache miss penalty will be 2(T+k).

CPU: The MIPS computer employs a multi-stage pipeline processor in order to dynamically schedule instruction execution. There are generally four stages in the pipeline, namely, Instruction fetch (IF), instruction decoding and operand reading (ID), Execution (EX), and Write back (WB). The EX stage include both the ALU and data memory (MEM) access stages. There are four distinct ALU units for: one Integer arithmetic, one FP add/subtract, one FP multiplication, and one FP division. The integer unit (IU) always takes one cycle for the execution. The specification of whether the individual FP units are pipelined or not, as well as the latency of each unit are to be provided in an input file with the name "*config.txt*". An example configuration is shown in Figure 2, where both the FP adder and multiplier are pipelined, tacking 4 and 6 cycles, respectively. The FP division is not pipelined and needs 20 cycles to complete. Note that the number of cycles for the instruction fetch (IF) and data access (MEM) stages depends on the cache performance, e.g. miss or hit. For load and store instructions, the address is calculated by IU before the data cache is accessed. Meanwhile for the integer instructions, e.g., DADD, the instruction will spend only 1 cycle MEM stage, before advancing WB stage.

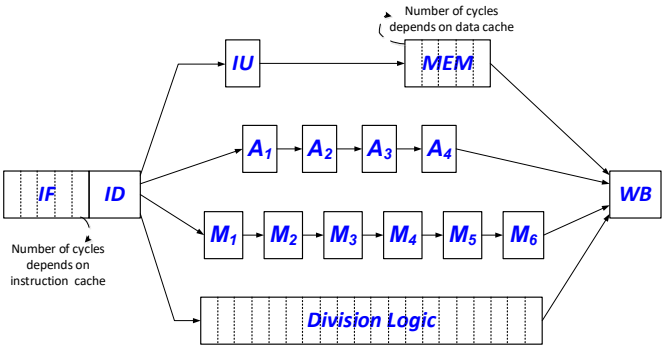

Figure 2: An example configuration for the execution stage

The pipelined processor enables instruction-level parallelism with in-order issue, out-of-order execution and out-of-order completion. All instructions should pass through the IF stage in order. If an instruction cannot leave the ID stage because a functional unit is busy, a subsequent instruction will not be fetched. However, if instructions use different functional units after they pass the ID stage, they can get stalled or bypass each other in the EX stage and thus leave the EX stage out of order. Unconditional jumps complete in the ID stage. The fetched instruction (the next instruction in the program) will be flushed from IF stage in that case. In other words, a "J" instruction will waste one cycle if the following instruction in the program is in cache, or waste as many cycles as the cache miss penalty if the next instruction was not in cache. Conditional branches are also resolved in the "ID" stage as well. Meanwhile the CPU will go ahead and fetch the next instruction, in other words, always "not-taken prediction" will be used in the IF stage. If the branch turns to be "non-taken" the pipeline will not be stalled. However, if the branch is taken, the control unit will flush the fetched instruction and update the program counter. In other words, if the branch is "Taken", one cycle will be wasted (unless a cache miss is encountered while the branch is being evaluated). The branching instructions do not stall because of structural hazard related to the integer unit. On the other hand, a conditional branching instruction may suffer RAW hazard, in which case the instruction will be stalled in the ID stage until the RAW hazard is resolved. The table below shows the number of cycles each instruction takes in the EX stage.

Instructions	Number of Cycles in "Execute" Stage
HLT, J	0 Cycles (finish in ID stage)
BEQ, BNE	0 Cycle (finish in ID stage)
DADD, DADDI, DSUB, DSUBI, AND, ANDI, OR, ORI	2 Cycles (one for IU + one for MEM stage)
LW, SW, L.D, S.D	1 Cycle + memory access time (D-Cache)
ADD.D, SUB.D	Specified in the "*config.txt*" file
MUL.D	Specified in the "*config.txt*" file
DIV.D	Specified in the "*config.txt*" file

As pointed out, the configuration of the FP add, multiply and division units is to be specified in one of the input files with the name as ***config.txt***. The simulator accepts three additional input files; one is for the program containing a mix of the assembly language instructions in Table 1, a file for the initial contents of data memory and another file for the values stored in the integer registers. The output of the simulator should contain the following information:

(i). The execution time a program takes (by reporting the cycle number that each instruction completes each stage it passes through).
(ii). The structural and data hazards (RAW, WAR and WAW) in the assembly code which result in pipeline stalls.
(iii). The performance of the instruction and data caches. The evaluation criteria should be the total number of cache access requests and the number of the cache hits for the particular cache.

You need to take into account the following additional facts:

1) The pipeline processor does not have forwarding hardware.
2) In addition to the 32 word-size registers (for integers), there are 32 FP registers; each has 64 bits.
3) Floating point calculations will have no impact on the required output of your simulator. In fact, only the contents of the integer registers need to be read from the input file, and you do not even need to allocate storage in your simulator for floating point registers.
4) The number of cycles required by the ALU depends on the latency of the involved functional unit and whether it is pipelined or not.
5) Instructions and data are stored in memory starting at address 0x0 and 0x100 respectively. Load and store instructions use word-aligned addresses when accessing data.
6) Both conditional and unconditional jump instructions can be forward and backward. You can assume that a program will not create a closed loop.
7) Integer and floating point operations use the same write port and hence structural hazards can occur. Structural hazards are detected before entering the WB stages. The functional unit that has the instruction will be stalled (stay busy) if the instruction cannot proceed to the WB stage. In case multiple instructions are ready at the same time to the WB stage, the priority will be given to the functional unit that in not pipelined and takes the most execution cycle (based on the parameters in "config.txt"). If there is a tie, the instruction that was issued the earliest will have the priority.
8) An instruction stalled for RAW hazard in the ID stage can get the values in the same cycle WB takes place.
9) WAW hazards are detected at the ID stages and resolved by stalling the pipeline.
10) All caches are blocking and do not support "hit under misses".
11) 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 decode stage. You can ignore the second HLT instruction.

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.

Your program must accept the input file in the format specified in the "simulator interface" section. The format of the output must **fully** comply with the specifications.

Simulator Interface

☺ ***Input files:*** There should be *FOUR* separate input files:

- The first input file is the instruction file "*inst.txt*" that contains the assembly language code, represented as a sequence of instructions in symbolic format such as *L.D* or *ADD.D*, based on the subset of MIPS instructions specified in Table 1 above. 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 second input 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.
- The third input file specifies the initial state of the processor's internal registers. There should be a sequence of 32 32-bit numbers to be assigned to the integer registers. These values capture the machine state at the time the assembly code is executed. The first data word in the file should be loaded into register R_0, the next into register R_1, and so on. There is no need to preload (or even track) the contents of F_0, ... , F_{31}.
- The fourth input file states the configuration of the CPU, which includes the names of the functional units and the number of needed cycles to complete the operation, the access time of the instruction and data caches as well as main memory. Only one integer unit is allowed in any configuration, thus there is no need to specify it in the "config.txt" file.
- The "config.txt" file should include:

  FP adder: <cycle count>, <pipelined: yes/no>
  FP Multiplier: < cycle count >, < pipelined: yes/no>
  FP divider: < cycle count >, < pipelined: yes/no>
  Main memory: <access time (number of cycles)>
  I-Cache: <access time (number of cycles)>
  D-Cache: <access time (number of cycles)>

  Access time is specified in terms of the number of cycles.

- All of the four input files and one output file should be specified in the format of *COMMAND-LINE ARGUMENTS.* For instance, a command-line input that starts the simulation program should look like:

  simulator inst.txt data.txt reg.txt config.txt result.txt

- The simulator should print the following statement and exit whenever the number of command-line arguments supplied differs from *Five*. For example:

```
linux2[1]% ./simulator
Usage: simulator inst.txt data.txt reg.txt config.txt result.txt
```

  Note: the role of input files is 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).

- General comments about the input files format:

- We are not going to test your 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., DADD, dadd, dAdd, DAdd, … are the same)
- You should strongly stick to the format of MIPS instructions. For example if your implementation assume the "store word" instruction as, "SW 32(R2), R1", it will be wrong (the correct format is SW R1, 32(R2)). The table at the end of this document specifies the format of all MIPS instructions covered in this project.

☺ ***Output file format:*** The simulator must output all desired information to *ONE* output file "*result.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 existence of RAW, WAR and WAW data hazards that cause STALLS. If an instruction is stalled because such a hazard, mark a "Y" in the corresponding entry, otherwise mark an "N". Please note that an instruction may be stalled because of multiple data hazards.
- The existence of structural hazard. If an instruction suffers stalls due to structural hazard, mark a "Y" in the corresponding entry; otherwise mark an "N".
- 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 below as your guideline for output file format. Do not put extra information such as your name in your output file.

Example:

The following are sample scenarios for which the input files and the expected output are shown.

inst.txt

```
GG:   L.D  F1,  4(R4)
      L.D  F2,  8(R5)
      ADD.D  F4,  F6,  F2
      SUB.D  F5,  F7,  F1
      MUL.D  F6,  F1, F5
      ADD.D  F7,  F2,  F6
      ADD.D  F6,  F1,  F7
      DADDI  R4,  R4,  4
      DADDI  R5,  R5,  4
      DSUB   R1,  R1,  R2
      BNE    R1,  R3,  GG
      HLT
      HLT
```

config.txt

```
FP adder:  4, yes
FP Multiplier:  6, yes
FP divider: 20, no
Main memory: 2
I-Cache: 1
D-Cache: 1
```

data.txt

00000000000000000000000000000010
00000000000000000010000101010101
00000000001001110101010100010100
00000101010111110001010101010101
00000000101010101010101010101111
00000100010010010101110000001010
00000000000000001111111111111111
00000000000000001111111111111111
00000000000000001111100010100110
00000000000000001010101101110110
00000000001010110100111101010111
00000000001000000000000000000111
00000000000000101110000000010101
00000000000000000010101110101101
00000000000000000000000000000000
00000000000001101101010011110101
00000000000000000000000001100001
00000000000000110000101010101001
00000000000000000000010101010101
00000000000000001111111111111000
00000000000000000000000000000011
00000000000000000000000000000001
00000000000000001111100110100110
00000000000000000010101010101010
00000000000001101101010011110101
00000000000000100001010111001010101
00000000000001101101111001010101
00000000000000000010101010111010
00000000000001001010101011110101
00000000000000000000000000010101
00000000000111101110000000000000
00000000000001101101010011110100

reg.txt

00000000001000000110011000000011
00000000000000000000000000000011
00000000000000000000000000000001
00000000000000000000000000000001
00000000000000000000000100000000
00000000000000000000000100000000
00000000000000111011111111010101
00000000000000000001111100100111
00000000000000010101010111010011
00000000000000000101010000000010
00000000000101011111101010101010
00000101011011110101000000000101
01101010101010101011010010110101
00000010101011110001101010101010
00000110101011000010101010101010
00000010101111101101010100011111
00000000000001010101010101101011
00000000000000000000111010101110
00000000000000000000000010101001
00000000000000001110101010101010
01010101011111010101010101011011
00000000000101010101101110101011
00000001110101010101010101011011
00000000000001110101010101010101
00000011010101010100000001010101
00000000000000001010101010010011
00110010010010110101010101001000
01101010101010101011101010101011
00000000101010101010111111101111
00000000000000000000010110101010
00000000010101010101010101010111
00000000000000000000000000000000

result.txt

Instruction	FT	ID	EX	WB	RAW	WAR	WAW	Struct
GG: L.D F1, 4(R4)	6	7	15	16	N	N	N	N
L.D F2, 8(R5)	7	8	17	18	N	N	N	Y
ADD.D F4, F6, F2	8	18	22	23	Y	N	N	N
SUB.D F5, F7, F1	18	19	23	24	N	N	N	N
MUL.D F6, F1, F5	24	25	31	32	N	N	N	N
ADD.D F7, F2, F6	25	32	36	37	Y	N	N	N
ADD.D F6, F1, F7	32	37	41	42	Y	N	N	N
DADDI R4, R4, 4	37	38	40	41	N	N	N	N
DADDI R5, R5, 4	43	44	46	47	N	N	N	N
DSUB R1, R1, R2	44	45	47	48	N	N	N	N

	BNE R1, R3, GG	45	48			Y	N	N	N
	HLT	48							
GG:	L.D F1, 4(R4)	49	50	53	54	N	N	N	N
	L.D F2, 8(R5)	50	51	60	61	N	N	N	Y
	ADD.D F4, F6, F2	51	61	65	66	Y	N	N	N
	SUB.D F5, F7, F1	61	62	66	67	N	N	N	N
	MUL.D F6, F1, F5	62	67	73	74	Y	N	N	N
	ADD.D F7, F2, F6	67	74	78	79	Y	N	N	N
	ADD.D F6, F1, F7	74	79	83	84	Y	N	N	N
	DADDI R4, R4, 4	79	80	82	83	N	N	N	N
	DADDI R5, R5, 4	80	81	84	85	N	N	N	Y
	DSUB R1, R1, R2	81	82	85	86	N	N	N	Y
	BNE R1, R3, GG	82	86			Y	N	N	N
	HLT	86	87			N	N	N	N
	HLT	92							

Total number of access requests for instruction cache: 25

Number of instruction cache hits: 21

Total number of access requests for data cache: 8

Number of data cache hits: 6

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 611, Fall 2019, 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 the file ***simulator*** was deleted from the directory.

In order to submit your work:

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.
3- Rename your project folder to your UMBC user name, e.g. abcde12.

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 abcde12.zip)
6- Submit your zip file (abcde12.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, 32(R2)	Load word to an integer register	Regs[R1]<- Mem[32+Regs[R2]]
SW R3, 16(R4)	Store word from an integer register	Mem[16+regs[R4]]<-Regs[R3]
L.D F1, 32(R2)	Load double word to floating point register	Regs[F1]<- Mem[32+Regs[R2]]
S.D F3, 24(R4)	Store double word	Mem[24+regs[R4]]<-Regs[F3]
DADD R1, R2, R3	Add signed	Regs[R1] <- Regs[R2] + Regs[R3]
DADDI R1, R2, 43	Add immediate signed	Regs[R1] <- Regs[R2] + 43
DSUB R1, R2, R3	Subtract signed	Regs[R1] <- Regs[R2] - Regs[R3]
DSUBI R1, R2, 43	Subtract immediate signed	Regs[R1] <- Regs[R2] - 43
AND R1, R2, R3	Bitwise AND	Regs[R1] <- Regs[R2] & Regs[R3]
ANDI R1, R2, 43	Bitwise AND-immediate	Regs[R1] <- Regs[R2] & 3
OR R1, R2, R3	Bitwise OR	Regs[R1] <- Regs[R2] | Regs[R3]
ORI R1, R2, 43	Bitwise OR-immediate	Regs[R1] <- Regs[R2] | 43
ADD.D F1, F2, F3	Add double word	Regs[F1] <- Regs[F2] + Regs[F3]
SUB.D F1, F2, F3	Subtract double word	Regs[F1] <- Regs[F2] - Regs[F3]
MUL.D F1, F2, F3	Multiply double word	Regs[F1] <- Regs[F2] * Regs[F3]
DIV.D F1, F2, F3	Divide double word	Regs[F1] <- Regs[F2] / Regs[F3]
J LABEL	Unconditional jump	PC <-LABEL
BNE R3, R4, Label	Branch not equal	If(R3 != R4) PC<-Label
BEQ R3, R4, Label	Branch equal	If(R3 == R4) PC<-Label
HLT	Stop fetching new instructions	