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

CMSC 611 – Advanced Computer Architecture

Class Project Due: *May 9th, 2017*

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 the *CDC 6600 scoreboard* scheme to dynamically schedule instruction execution and employ 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, LI, LUI 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 as an input a program 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:

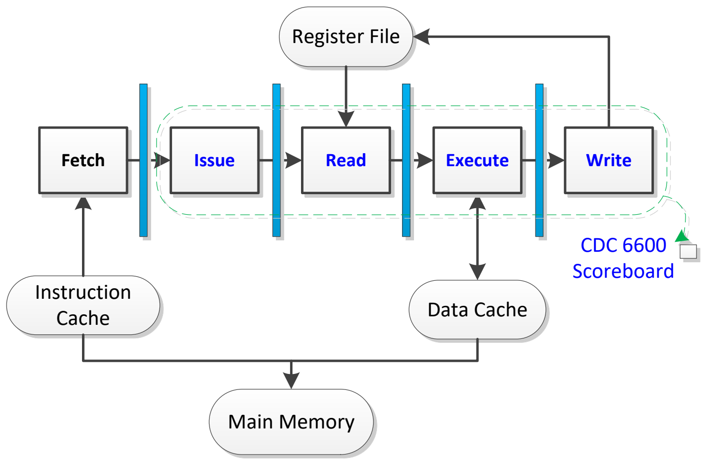

Figure 1: Block diagram description of a MIPS Computer

Memory: The MIPS machine is assumed to have an instruction cache (I-Cache) with an access (hit) time of one cycle per word. The organization of I-Cache is direct-mapped. The number of blocks and the block size in words are to be provided as in an input to the simulation in a configuration file. In addition, the machine has a data cache (D-Cache) with hit time of one cycle per word. 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. A write-back strategy is employed with a write-allocate policy. 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 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. The main memory is accessible through one-word-wide bus, and its access time is 3 cycles per word. If both caches experience a miss at the same cycle, the I-Cache will be given priority.

CPU: The MIPS computer employs the CDC 6600 scoreboard scheme in order to dynamically schedule instruction execution. An example organization of a MIPS processor with a scoreboard, 2 FL multipliers, 1 FL Divider, 1 FL Adder and an integer ALU, is shown in Figure 2. There are four stages in the scoreboard-based pipeline: Issue, Read operands, Execution, and Write results, which replace the traditional ID, EX and WB stages in MIPS. The scoreboard mechanism can effectively achieve instruction-level parallelism with in-order issue, out-of-order execution and out-of-order completion. All instructions pass through the issue stage in order (in-order issue); however, they can get stalled or bypass each other in the second stage (read operands) and thus enter execution out of order. *The read operand stage includes buffers to copy register values; thus there is no WAR hazard experienced. In addition, data transfer instructions (L.D, S.D, LW and SW) do not use the integer unit. Thus, there is no structural hazard will be experienced if an integer instruction is being executed and a load/store instruction is to be issued.* When a fetched instruction cannot be issued, the next instruction cannot be fetched. An instruction cache miss will add the I-Cache miss penalty to the fetch time.

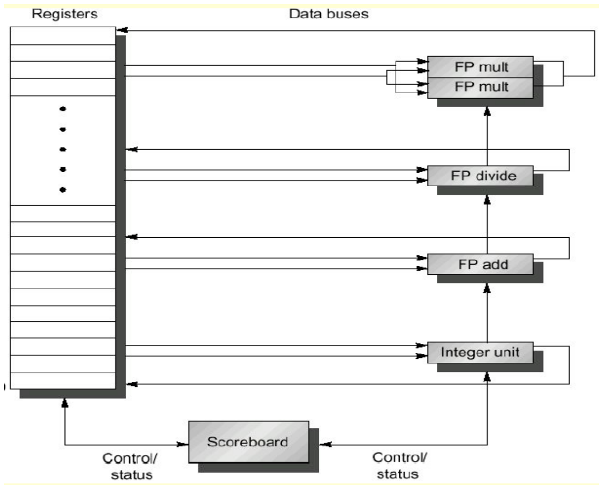

Figure 2: The basic organization of a MIPS processor with a scoreboard

Unconditional jumps complete in the "Issue" stage. The fetched instruction (the next instruction in the program) will be flushed from the "Fetch" stage in that case. In other words a "J" instruction will waste one cycle. On the other hand, conditional branches are resolved in the "Read" stage. Meanwhile the CPU will go ahead and fetch the next instruction, in other words, always "not-taken prediction" will be used in the "Fetch" stage. When a "BEQ" or "BNE" instruction is issued, the pipeline will be stalled until the condition is resolved. In other words, if the branch is "non-taken" one cycle will be wasted. Meanwhile, if we find out in the "Read" stage that the branch is taken, the control unit will flush the "Fetch" stage and update the program counter so the CPU in next cycle will fetch from the branch target address. In other words, if the branch is "Taken", 2 cycles will be wasted. *Please note that if a "Fetch" causes an I-Cache miss, the "Fetch" is to be completed before it is flushed when the branch is "Taken".* The branching instructions do not stall because of structural hazard related to the integer unit.

The table below shows the number of cycles each instruction takes in the "Execute" stage. Please keep in mind that an LI instruction will pass through the integer unit, even though there is no calculation to be made. Thus, an LI instruction will prevent the issue of another instruction that uses the integer unit. Also, the LI instruction supports only 16-bit constants. If a 32-bit constant is to be loaded into a register, a sequence of "LUI" and "ORI" is to be used.

Instructions	Number of Cycles in "Execute" Stage
HLT, J	0 Cycle (finish in Issue stage)
BEQ, BNE	0 Cycle (finish in Read stage)
DADD, DADDI, DSUB, DSUBI, AND, ANDI, OR, ORI, LI, LUI	1 Cycle
LW, SW	1 Cycle + D-Cache miss penalty if applicable
L.D, S.D	2 Cycles + D-Cache miss penalty if applicable
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

The number of functional units is to be specified in one of the input files with the name as ***config.txt***. The simulator accepts two additional input files; one is for the program containing a mix of the assembly language instructions in Table 1, and a file for the initial contents of data memory. The output of the simulator should contain the following information:

(i). Structural and data hazards (RAW and WAW) in the assembly code which result in pipeline stalls.

(ii). The execution time a program takes (by reporting the cycle number that each instruction completes each stage it passes through).

(iii). The performance of the data and instruction 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) In a MIPS processor with scoreboard, there is no 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, 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.
5) The register file has multiple write ports; thus, no structural hazard will occur if multiple instructions reach the write stage at the same time.
6) Instructions and data are stored in memory starting at address 0x0 and 0x100 respectively. Load and store instructions uses word-aligned addresses when accessing data.
7) Both conditional and unconditional jump instructions can be forward and backward. You can assume that a program will not create a closed loop.
8) All caches are blocking and do not support "hit under misses".
9) The HLT instruction will mark the end of the program, i.e., fetching will seize as soon as the HLT instruction is issued. 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 issue stage. You can ignore the second HLT instruction and do not perform a cache miss.

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. You also MUST include execution syntax in README file, e.g., "java simulator inputFile.asm data.txt output.txt". In addition, the README file should provide a brief explanation of submission package (files) and software environment. 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 *THREE* 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. There also 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 states the configuration of the scoreboard, which includes the number of functional units in the scoreboard and the number of needed cycles to complete the operation. Only one ***Integer*** unit is allowed in any configuration (and thus there is no need to specify it in the "confi.txt" file). The instruction cache is always direct-mapped; the configuration file will specific the ***I-Cache*** size in terms of the number of blocks and the size of a block. Thus, the "config.txt" file should include:

  FP adder: <number of units>, <cycle count>
  FP Multiplier: <number of units>, <cycle count>
  FP divider: <number of units>, <cycle count>
  I-Cache: <number of blocks>, <block size in words>
- All of the three 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 config.txt result.txt
- The simulator should print the following statement and exit whenever the number of command-line arguments supplied differs from *FOUR*. For example:

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

  Note: the role of <u>input files</u> 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 you implement the load immediate as, "LI 1, R1", it will be wrong (the correct format is "LI R1, 1"). 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 <u>leaves</u> every stage. If an instruction does not enter a particular stage, leave the entry empty. For example, the "BNE" instruction finishes in the "Read" stage and thus there will be no entries in the following stages for this instruction.
- The existence of RAW 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.

Config.txt

```
FP adder:  2, 2
FP Multiplier:  2, 30
FP divider: 1, 50
I-Cache: 4, 4
```

data.txt

```
00000000000000000000000000000010
00000000000000000010000101010101
00000000001001110101010100010100
00000101010111110001010101010101
00000000101010101010101010101111
00000100010010010101110000001010
00000000000000001111111111111111
00000000000000001111111111111111
00000000000000001111100010100110
00000000000000001010101101110110
00000000000101011010011110101011
00000000000100000000000000000111
00000000000000101110000000010101
00000000000000000010101110101101
00000000000000000000000000000000
00000000000001101101010011110101
00000000000000000000000001100001
00000000000000110000101010101001
00000000000000000000010101010101
00000000000000001111111111111000
00000000000000000000000000000011
00000000000000000000000000000001
00000000000000001111100110100110
00000000000000000001010101010101
00000000000001101101010011110101
00000000000001000010101110010101
00000000000001101101111001010101
00000000000000000101010101110100
00000000000001001010101011110101
00000000000000000000000000010101
00000000000111011100000000000000
00000000000001101101010011110100
```

inst.txt

```
         LI  R4,    266
         LI  R5,    274
         LI  R1,    8
         LI  R2,    4
         LI  R3,    0
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,  2
         DADDI   R5,  R5,  2
         DSUB    R1,  R1,  R2
         BNE     R1,  R3,  GG
         HLT
         HLT
```

result.txt

Instruction	Fetch	Issue	Read	Exec	Write	RAW	WAW	Struct
LI R4, 266	13	14	15	16	17	N	N	N
LI R5, 274	14	18	19	20	21	N	N	Y
LI R1, 8	18	22	23	24	25	N	N	Y
LI R2, 4	22	26	27	28	29	N	N	Y
LI R3, 0	35	36	37	38	39	N	N	N
GG: L.D F1, 4(R4)	36	37	38	52	53	N	N	N
L.D F2, 8(R5)	37	54	55	80	81	N	N	Y
ADD.D F4, F6, F2	54	55	82	84	85	Y	N	N
SUB.D F5, F7, F1	67	68	69	71	72	N	N	N
MUL.D F6, F1, F5	68	69	73	103	104	N	N	N
ADD.D F7, F2, F6	69	73	105	107	108	Y	N	Y
ADD.D F6, F1, F7	73	105	109	111	112	Y	Y	Y
DADDI R4, R4, 2	105	106	107	108	109	N	N	N
DADDI R5, R5, 2	106	110	111	112	113	N	N	Y
DSUB R1, R1, R2	110	114	115	116	117	N	N	Y
BNE R1, R3, GG	114	115	118			Y	N	N
HLT	127					N	N	N
GG: L.D F1, 4(R4)	128	129	130	144	145	N	N	N
L.D F2, 8(R5)	129	146	147	149	150	N	N	Y
ADD.D F4, F6, F2	146	147	151	153	154	Y	N	N
SUB.D F5, F7, F1	147	148	149	151	152	N	N	N
MUL.D F6, F1, F5	148	149	153	183	184	Y	N	N
ADD.D F7, F2, F6	149	153	185	187	188	Y	N	Y
ADD.D F6, F1, F7	153	185	189	191	192	Y	Y	Y
DADDI R4, R4, 2	185	186	187	188	189	N	N	N
DADDI R5, R5, 2	186	190	191	192	193	N	N	Y
DSUB R1, R1, R2	190	194	195	196	197	N	N	Y
BNE R1, R3, GG	194	195	198			Y	N	N
HLT	195	199				N	N	N
HLT	199					N	N	N

Total number of access requests for instruction cache: 30

Number of instruction cache hits: 25

Total number of access requests for data cache: 8

Number of data cache hits: 2

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, Spring 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 the file ***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 is acceptable. (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
LI R8, 42	Load immediate	Regs[R8] <- 42
LI R8, -42	Load immediate *	Regs[R8] <- (-42)
LUI R5, 42	Load a value into the most significant 16 bits	Regs[R8] <- $2^{16} \times 42$
LW R1, 30(R2)	Load word to an integer register	Regs[R1]<- Mem[30+Regs[R2]]
SW R3, 500(R4)	Store word from an integer register	Mem[500+regs[R4]]<-Regs[R3]
L.D F1, 30(R2)	Load double word to floating point register	Regs[F1]<- Mem[30+Regs[R2]]
S.D F3, 500(R4)	Store double word	Mem[500+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	

* *Immediate values can be positive or negative*