CSCE 513 Computer Architecture, Fall 2018, Assignment #2, due 10/08/2018, 11:55PM

Covered topics: 1) pipeline, hazards, and instruction scheduling. 2) pipeline implementation. 3) Cache Organization and Cache Performance.
Total points: 100+40 (bonus) points for undergraduates, and 130+10 (bonus) for graduates.

Problem	1	2	3	4*	5.1	5.2	5.3
Points	40+10	5	5	30*	10	20	20

- **Question 1.1 has bonus 10 points.**
- **Question 4 is required for graduate students. Undergraduates who solve receive bonus points.**
- **For questions 1 and 4, prepare your solutions using the provided Excel sheets. When you complete, copy the required tables and figures to the PDF file of your submission.**

Submission:

1. **Only electronic submissions on dropbox are accepted.**
2. **All your solutions should be included in a SINGLE PDF file.**
3. **Number your solutions in the same way and in the same order as the questions are numbered in this document and do NOT include the questions as part of your submission.**
4. **Include your full name in the PDF file.**
5. **Scanned copy of handwritten answers will NOT be graded.**

Problem 1. (Total 40+10 points. 1: 10+10 points, 2: 20 points, 3: 10 points)

The following code is compiled to RISC-V processors implemented in 64-bit 5-stage pipeline. In this assignment, you need to work on the following three questions and an Excel file that contains three sheets is provided for you to do that:
1) Fill in the table in Sheet "1.1 Information" with the information for each instruction (the encoding column is bonus for 10 points). Include the table in the PDF file of your submission.
2) In each of the following 3 configurations, draw the pipeline execution graph using Sheet "1.2 Pipeline and Cycle Counting" to show the scheduling of the instructions in each cycle, assuming N = 1000. Shorten your drawing as you see fit (e.g. you do not need to draw each of the 1000 iterations). Include the table from the Excel file in the PDF file of your submission.

The three configurations:

1. No structure hazards, no register forwarding or any support for dealing with hazards of control transfer instructions
2. No structure hazards, register forwarding, but no support for dealing with hazards of control transfer instructions

3. No structure hazards, register forwarding, improved ID stage for branch test and computing new PC (stall cycles reduced from 3 to 1).

3) In each of the 3 configurations, count the total number of cycles and the amount of stall cycles for executing the loop of the program, the green-highlighted portion. Put the total cycles and total stall cycles of all the 3 configurations in Sheet "1.3 Cycle Counting and Plot" to show the differences pictorially of these two metrics of the 3 configurations. Explain the table using a short paragraph (less than ¼ pages). Include the "Cycle Counting Plot" figure and your explanation in the PDF file of your submission.

Original source code:

```
int sum(int N, int a, int *X) {
    int i;
    int result = 0;
    for (i = 0; i < N; ++i)
        result += X[i];
    return result;
}
```

RISC-V Assembly code compiled into 64-bit ISA with comments added manually:

```
        .file   "sum.c"
        .text
        .align  2
        .globl  sum
        .type   sum, @function
sum:
        add     sp,sp,-48     /* update the stack pointer for this function */
        sd      s0,40(sp)     /* push the caller frame pointer to the stack */
        add     s0,sp,48      /* update the frame pointer for this function */
        sw      a0,-36(s0)    /* store N in the current frame */
        sw      a1,-40(s0)    /* store a in the current frame */
        sd      a2,-48(s0)    /* store int * X in the current frame */
        sw      zero,-24(s0)  /* int result = 0 */
        sw      zero,-20(s0)  /* int i = 0 */
        j       .L2           /* local jump to .L2 */
.L3:
        lw      a5,-20(s0)    /* a5 = i */
        sll     a5,a5,2       /* a5 = i<<2, which is i=i*4 */
        ld      a4,-48(s0)    /* a4 = X */
        add     a5,a4,a5      /* the &X[i] */
        lw      a5,0(a5)      /* the X[i] */
        lw      a4,-24(s0)    /* load result */
        addw    a5,a4,a5      /* result += X[i] */
```

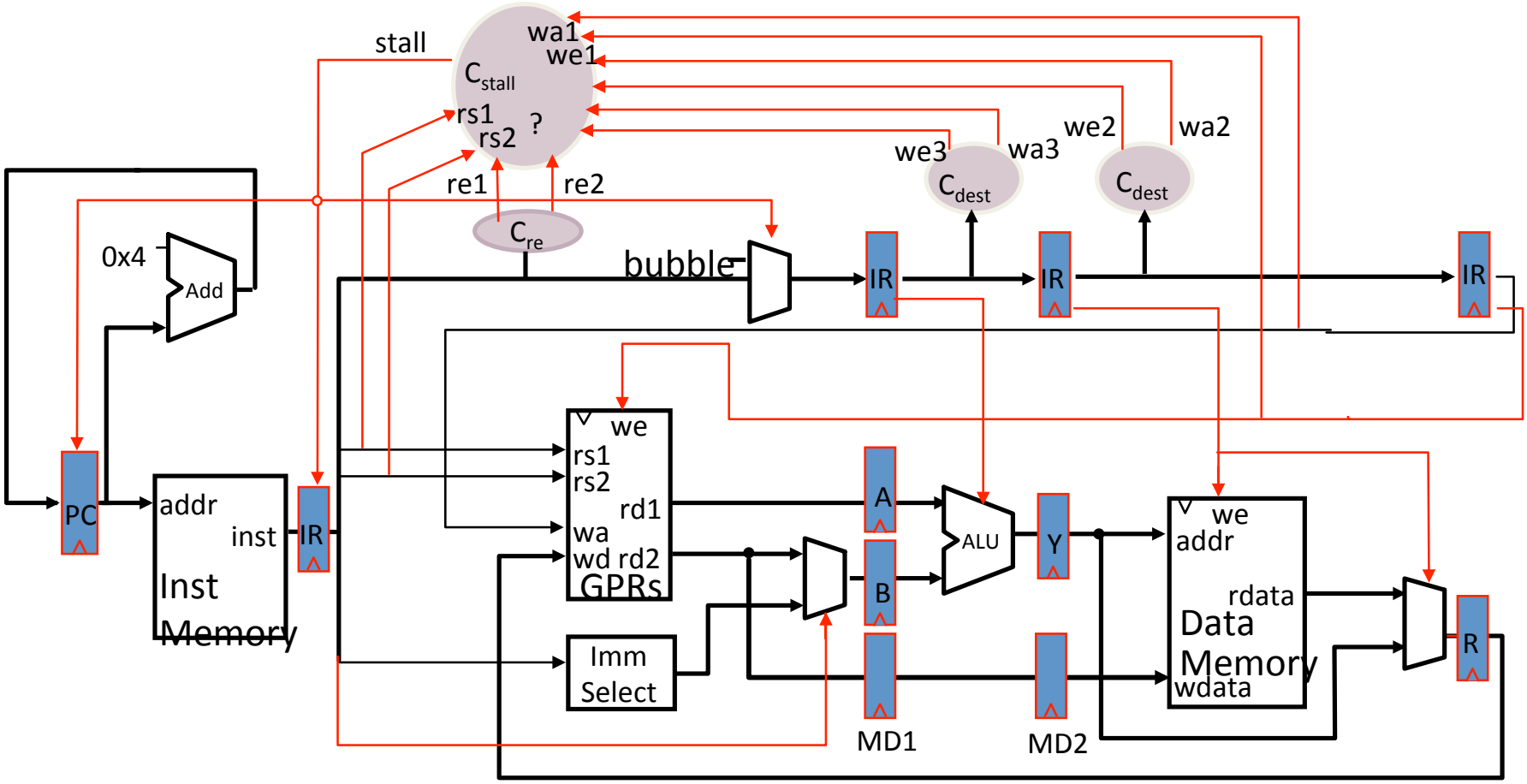
```
        sw      a5,-24(s0)    /* store to result */
        lw      a5,-20(s0)    /* i */
        addw    a5,a5,1       /* i++ */
        sw      a5,-20(s0)    /* store i */
.L2:
        lw      a4,-20(s0)    /* i */
        lw      a5,-36(s0)    /* N */
        blt     a4,a5,.L3     /* if (i < N) goto .L3 */
        lw      a5,-24(s0)    /* load result */
        mv      a0,a5         /* the register for the return value */
        ld      s0,40(sp)     /* reset the frame pointer (fp) to the caller */
        add     sp,sp,48      /* restore the stack pointer (sp) for the caller */
        jr      ra            /* jump back to the caller, ra: return address */
       .size   sum, .-sum
       .ident  "GCC: (GNU) 6.1.0"
```

The following table shows the register usage for function call, e.g. ra is x1, sp is x2, and fp is x8. Please refer to the riscv.org spec for details about the encoding (https://riscv.org/specifications/, Chapter 19). For encoding, the assembly codes use instruction opcode mnemonic which may not fully tell the type of the instruction. E.g. add sp sp -48 is actually an addi, I-type since it has an immediate in the instruction. You should encode it as addi instruction instead of a R-type add instruction.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

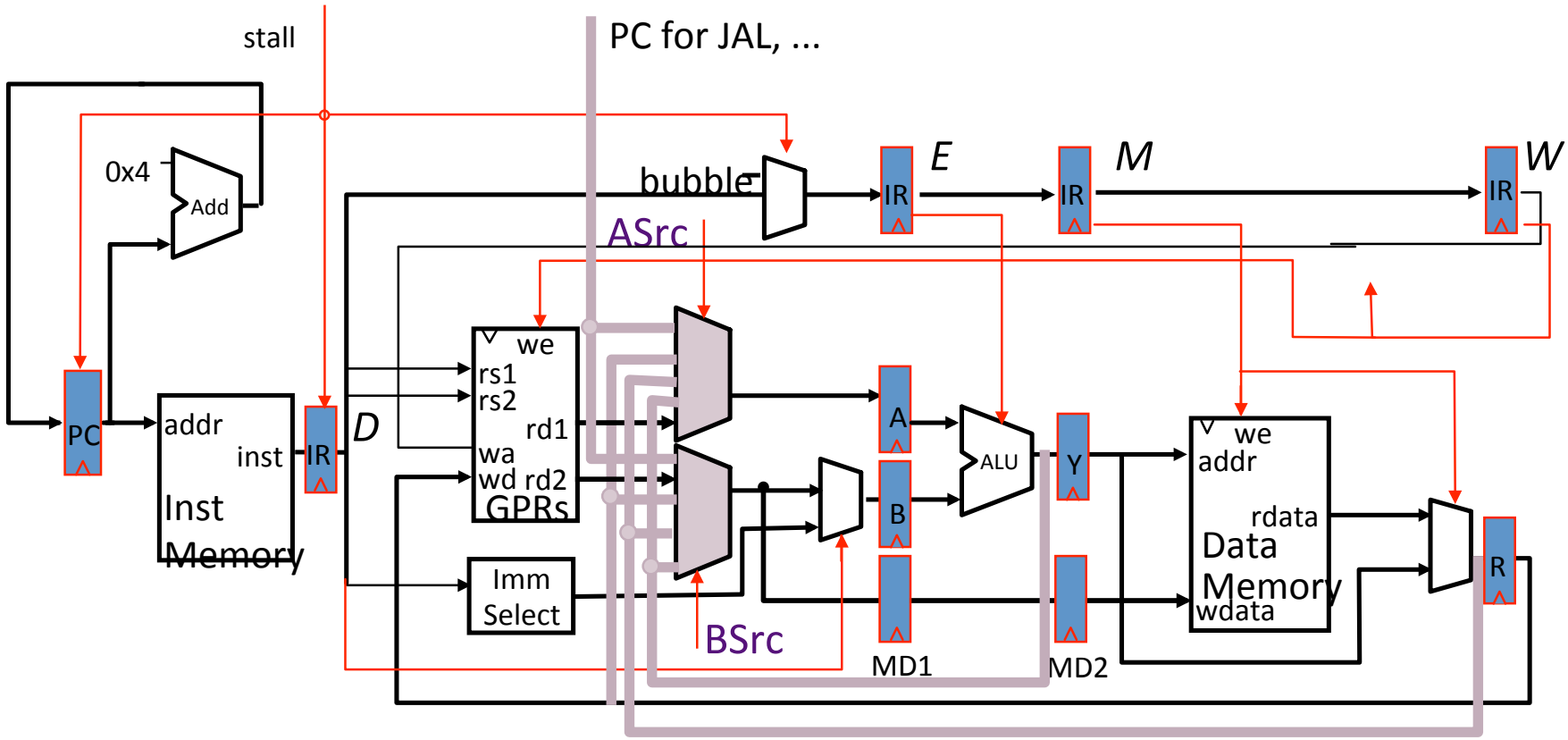
Table 20.2: RISC-V calling convention register usage.

Problem 2. (Total 5 points)



The above figure shows the interlock control logic for handling RAW data hazards by inserting bubbles. List the control signals used for handling RAW hazards between two instructions in ID-EXE, ID-MEM and ID-WB stages. For example, for two instructions in ID-EXE stages, we will need rs1, rs2, re1, re2, we3 and wa3 signals.

Problem 3. (Total 5 points)



The above figure shows the data path for handling RAW data hazards by bypassing. Label the data path used for handling RAW hazards between two instructions in EXE-EXE, MEM-EXE and WB-EXE stages using EXE-EXE, MEM-EXE and WB-EXE

Problem 4. (Total 30 points, required for graduate and bonus for undergraduate)

In this problem, you will modify a RISC-V 32-bit 5-stage pipeline design (UCB Sodor core) to experiment the impact of bypassing for handling data hazards. The design is implemented in Chisel and your modification will change the pipeline data and control path so you can experiment 4 bypassing configurations: fully-interlocked (no bypassing), EXE->EXE(forward output of EXE to the input of EXE), EXE|MEM->EXE (forward output of EXE and MEM stage to the input of EXE), and EXE|MEM|WB->EXE(full-bypassing).

The latest release of Sodor core (https://github.com/ucb-bar/riscv-sodor) does not produce a summary of instruction tracing we need when being simulated, thus we will use an older version of the Sodor core (https://github.com/passlab/riscv-sodor). This https://github.com/ucb-bar/riscv-sodor/wiki provides a high-level overview of the Sodor core.

Instructions to set up the environment on a CSCE linux machine for the design, simulation, and experiment.
The procedure and commands were performed on Linux machines in Swearingen 1D43 and 3D22. I believe it works on other Linux machines as well.

1. Login to one of the machine using ssh/putty. Check https://passlab.github.io/CSCE513/resources/devmachine.html for details.

2. Download the Sodor source folder to your home folder using the following command (**only do once**)

```
-bash-4.1$ cd ~
-bash-4.1$ wget https://passlab.github.io/CSCE513/Assignment_2/Assignment_2-riscv.tar
-bash-4.1$ tar xf Assignment_2-riscv.tar
```

Part of the `riscv/riscv-sodor` source code directory structure is shown as:

- `Makefile`
- `src/` `/* Chisel source code for each RISC-V implementation */`
  - `common` `/* common source codes shared between all processors */`
  - `rv32_1stage` `/* source code for single-cycle RISC-V processor */`
  - `rv32_2stage`
  - `rv32_3stage`
  - `rv32_5stage` `/* source code for single-cycle RISC-V processor */`
  - `rv32_ucode`
- `emulator/` `/* C++ emulator source code */`
  - `common` `/* common emulation infrastructure for all processors */`
  - `rv32_1stage` `/* C++ emulation code for single-cycle RISC-V processor */`
  - `rv32_2stage`
  - `rv32_3stage`
  - `rv32_5stage` `/* C++ emulation code for 5-stage pipeline RISC-V processor */`
  - `rv32_ucode`

3. Configure your build environment for the source code (**only do once**)

```
-bash-4.1$ cd riscv/riscv-sodor
-bash-4.1$ ./configure --with-riscv=$HOME/riscv/local
```

4. Build and run the simulator (**need to run the command each time you change the source code in `src/rv32_5stage`**)

```
-bash-4.1$ cd emulator/rv32_5stage
-bash-4.1$ make run
```

If this is your first time, this command may take a while. Don't worry about *unconnected input/floating output* warnings. The command `make run` does the following:

- runs `sbt`, the Scala Built Tool, selects the `rv32_5stage` project, and runs the `Chisel` code which generates a C++ cycle-accurate description of the processor. The generated C++ code can be found in `emulator/rv32_5stage/generated-src/`
- compiles the generated C++ code into a binary called `emulator`.
- calls the RISC-V front-end server (called `fesvr`), which opens a socket to your C++ binary `emulator`, and sends it a RISC-V binary for the target processor to execute. All of the RISC-V tests and benchmarks will be executed when calling "`make run`".

A `PASSED` should be generated by each of the 6 programs, see below. If you see any `FAILED`, try again or contact me.

```
        ......
[ PASSED ] output/median.riscv.out
[ PASSED ] output/multiply.riscv.out
[ PASSED ] output/qsort.riscv.out
[ PASSED ] output/towers.riscv.out
[ PASSED ] output/vvadd.riscv.out
[ PASSED ] output/dhrystone.riscv.out

-bash-4.1$
```

Inspect the simulation output of the 6 programs: `median`, `multiply`, `qsort`, `towers`, `dhrystone`, and `vvadd`. Using your editor of choice, look at the output files generated from `make run`. The `more` command lists the contents from the beginning and you can hit blank key to roll forward, and q to quit the browsing. The "`tail -n 16`" command outputs the last 16 lines of the file, which show the statistics from tracing.

```
-bash-4.1$ more output/vvadd.riscv.out
-bash-4.1$ tail -n 16 output/vvadd.riscv.out

#---------- Tracer Data ----------
#
#      CPI   : 1.27
#      IPC   : 0.79
#      cycles: 3426
#
#      Bubbles     : 20.957 %
#      Nop instr   : 0.000 %
#      Arith instr : 37.916 %
#      Ld/St instr : 30.385 %
#      branch instr: 9.982 %
#      misc instr  : 0.759 %
#---------------------------------

*** PASSED ***
-bash-4.1$
```

The 5-stage processor has been parameterized to support both full-bypassing (but must still stall for load-use hazards) and fully-interlocked. The fully-interlocked variant provides no bypassing, and instead must stall (interlock) the IF and ID stages until all hazards have been resolved. To set the pipeline to "Fully-Bypassing" or "Fully-interlocked", navigate to the Chisel source code: `src/rv32_5stage/consts.scala` (https://github.com/passlab/riscv-sodor/blob/master/src/rv32_5stage/consts.scala). The file `consts.scala` provides constants and machine parameters for the processor. Change the parameter `USE_FULL_BYPASSING` to `true` or `false` to enable "Full-Bypassing" or "Fully-interlocked. How this parameter impacts the pipeline are shown in the data path in `dpath.scala` (https://github.com/passlab/riscv-sodor/blob/master/src/rv32_5stage/dpath.scala#L203, lines ~200-240) and the control path in `cpath.scala` (https://github.com/passlab/riscv-sodor/blob/master/src/rv32_5stage/cpath.scala#L222, lines ~220-250). The data path has bypass muxes used when full bypassing is activated. The control path contains the stall logic, which must be accounted for situations when no bypassing is supported. After you change the `USE_FULL_BYPASSING` parameter, follow the steps starting from #3 to run the simulator and inspect/record the tracing output of each program.

To experiment with the other two bypassing configurations, EXE->EXE and EXE|MEM->EXE, you will need to modify the `Chisel` source code found in `src/rv32_5stage`. The `dpath.scala` and `cpath.scala` files contain the relevant code for you to modify the bypass paths and stall logic. Make sure that your modified pipeline passes the assembly tests! After modification, follow the steps starting from #3 to run the simulator and inspect/record the tracing output of each program.

For each of the four bypassing configurations, you will collect CPI, cycles and bubble percentage for each of the 6 programs and put these results in the "Problem 4" sheet included in the Excel file. For each of the three metrics (CPI, cycles and bubble percentage), a plot figure is already created so after you input the results in the sheet table, the plot figure will be automatically populated. (The current numbers in the sheet are dummy numbers.)

In the submission of your solution for this problem, please include the following in the PDF file of your submission:

1. **Code snippet for the changes you made (not the whole file) and the filename.**
2. **The three plot figures.**
3. **Based on the results and figures, provide your quantitative explanation for the impact to CPI, total cycles and bubble percentage by each bypassing configuration and data path.**

Problem 5.1. (Total 10 points)

Assume memory is byte addressable and words are 64 bits, unless specified otherwise.

5.1 In this exercise we look at memory locality properties of matrix computation. The following code is written in C, where elements within the same row are stored contiguously. Assume each word is a 64-bit integer.

```
for (I=0; I<8; I++)
 for (J=0; J<8000; J++)
  A[I][J]=B[I][0]+A[J][I];
```

5.1.1 [5] <§5.1> How many 64-bit integers can be stored in a 16-byte cache block?

5.1.2 [5] <§5.1> Which variable references exhibit temporal locality?

5.1.3 [5] <§5.1> Which variable references exhibit spatial locality? Locality is affected by both the reference order and data layout.

Problem 5.2. (Total 20 points)

5.2 Caches are important to providing a high-performance memory hierarchy to processors. Below is a list of 32-bit memory address references, given as word addresses.

3, 180, 43, 2, 191, 88, 190, 14, 181, 44, 186, 253

5.2.1 [10] <§5.3> For each of these references, identify the binary address, the tag, and the index given a direct-mapped cache with 16 one-word blocks. Also list if each reference is a hit or a miss, assuming the cache is initially empty.

5.2.2 [10] <§5.3> For each of these references, identify the binary address, the tag, and the index given a direct-mapped cache with two-word blocks and a total size of 8 blocks. Also list if each reference is a hit or a miss, assuming the cache is initially empty.

5.2.3 [20] <§§5.3, 5.4> You are asked to optimize a cache design for the given references. There are three direct-mapped cache designs possible, all with a total of 8 words of data: C1 has 1-word blocks, C2 has 2-word blocks, and C3 has 4-word blocks. In terms of miss rate, which cache design is the best? If the miss stall time is 25 cycles, and C1 has an access time of 2 cycles, C2 takes 3 cycles, and C3 takes 5 cycles, which is the best cache design?

There are many different design parameters that are important to a cache's overall performance. Below are listed parameters for different direct-mapped cache designs.

Cache Data Size: 32 KiB

Cache Block Size: 2 words

Cache Access Time: 1 cycle

Problem 5.3. (Total 20 points)

5.3 For a direct-mapped cache design with a 32-bit address, the following bits of the address are used to access the cache.

Tag	Index	Offset
31–10	9–5	4–0

5.3.1 [5] <§5.3> What is the cache block size (in words)?

5.3.2 [5] <§5.3> How many entries does the cache have?

5.3.3 [5] <§5.3> What is the ratio between total bits required for such a cache implementation over the data storage bits?