

## CS-GY-6133 Homework 2

### Q1. Single-cycle Processor Design [25 Points]

In this problem, you will design the micro-architecture of a synchronous, single-cycle processor that supports the CSGY6133 instruction set architecture. The CSGY6133 has the following features:

- CSGY6133 has a register file (RF) with 8 general purpose registers. Each register holds a 32-bit value.
- CSGY6133 can address up to 4GB of Byte addressable main memory. (Therefore it has 32-bit addresses)
- CSGY6133 is a 2-address ISA. Each CSGY6133 instruction is 8 bits long and has the format shown below. The two MSBs are reserved for the op code. Each instruction has two operands, registers rs and rd. Since the RF has 8 registers, the address of each register is 3 bits long.

7	6 5	3 2	0
Opcode	Rd	Rs	

CSG6133 supports the following four instructions. The opcodes and semantics of each instruction are described below.

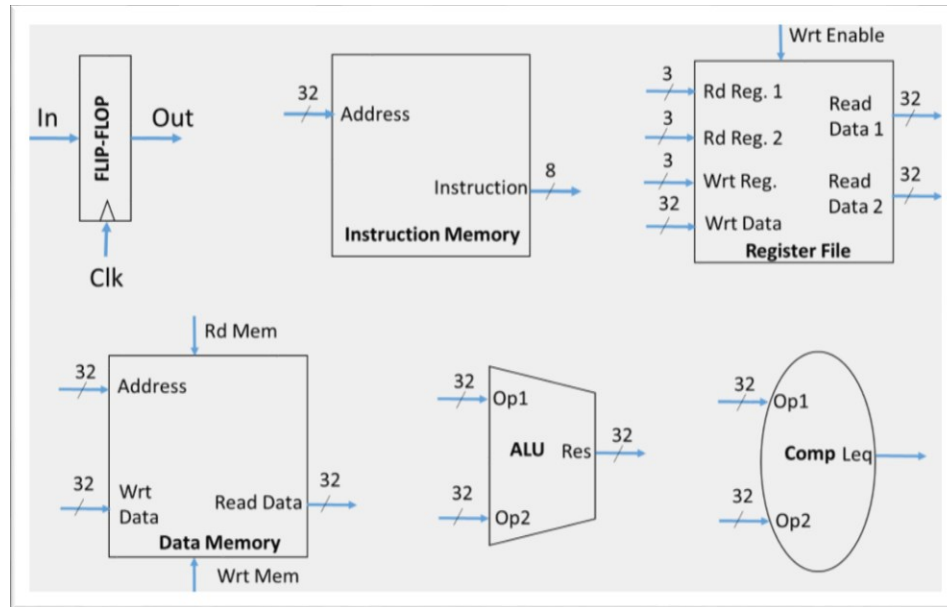
Instruction	Opcode	Semantics
sub rd, rs	00	$R[rs] \leftarrow R[rs] - R[rd]$ Subtract the value in register rd from the value in register rs. Store the result in register rs.
subm rd, rs	01	$R[rs] \leftarrow R[rs] - \text{Mem32}[R[rd]]$ Subtract the 32-bit value stored in memory starting at the address stored in register rd from the value in register rs. Store the result in register rs.
st rd, rs	10	$\text{Mem32}[R[rd]] \leftarrow R[rs]$ Store the value in register rs in memory starting at the address stored in register rd.
bleq rd, rs	11	If $R[rd] \leq 0$ then $PC = R[rs]$ else $PC = PC + 1$

\*\* Note that in the table above,  $\text{Mem32}[\text{Add}]$  takes as input a 32-bit address Add and returns a 32-bit value as follows:  $\text{Mem32}[\text{Add}] = \{\text{Mem}[\text{Add}+3], \text{Mem}[\text{Add}+2], \text{Mem}[\text{Add}+1], \text{Mem}[\text{Add}]\}$

where  $\text{Mem}[\text{Add}]$  is the 8-bit value stored in memory at address  $\text{Add}$ .

Your task is to design the micro-architecture of a processor that implements the CSGY6133 ISA. The processor is a synchronous single-cycle micro-architecture. In other words,

- It fetches and executes one instruction in every clock cycle.
- Updates to architectural state (PC, writes to register files and data memory) are synchronized to positive clock edges.



In your implementation, you are allowed to use the following hardware blocks.

1. Flip-flop: the flip-flop “samples” its input on every positive clock edge, and is stable otherwise.
2. Instruction memory (IM): the IM reads a 32-bit address and returns the corresponding 8-bit instruction at that address.
3. Register File (RF): the RF reads two 3-bit addresses, Rd Reg. 1 and Rd Reg. 2, and returns the 32-bit values stored in the respective registers. It also reads a 3-bit Wrt Reg. address and updates the corresponding register with 32-bit Wrt Data. If there is a write and read attempt to/from the same register, you can assume the write happens first. Finally, the Write Enable control signal must be set to 1 to enable the write operation.
4. ALU: reads two 32-bit values, Op 1 and Op2, and subtracts Op2 from Op1. The result is output to the 32-bit Res port.
5. Comparator (Comp): the comparator compares two 32-bit operands (Op1 and Op2) and outputs a 1 if  $\text{Op1} \leq \text{Op2}$ .
6. Data memory (DM): reads a 32-bit Address and if the Rd Mem. control signal is set to 1, outputs  $\text{Mem}_{32}[\text{Read Address}]$  (i.e., the 32-bits stored starting at Address). On the other

hand, if the Wrt Mem. control signal is set to 1, it writes the 32-bit value in Wrt Data to Mem32[Read Address] (i.e., to the 32-bits starting at Address). The Rd Mem and Wrt Mem control signals should never be set to 1 in the same cycle.

In addition, to the blocks above, you might need additional hardware blocks like MUXes, adders (to increment PC), and a decoder to generate control signals. You can specify how the decoder generates control signals using pseudo-code similar to what we did in class.

- 1.1 [5 Points] Draw a micro-architecture diagram of a single-cycle synchronous processor that supports only the *sub* instruction.
- 1.2 [5 Points] To the micro-architecture above, add support for the *subm* instruction. Your processor should support *sub* and *subm*.
- 1.3 [5 Points] Now, add support for the *st* instruction. Your processor should support *sub*, *subm* and *st*.
- 1.4 [5 Points] Finally, add support for the *bleq* instruction. This final installation should support the full CSGY6133 ISA.
- 1.5 [5 Points] Assume that every hardware block in your micro-architecture has a 1 ns latency. This includes the IM, DM, RF and ALU, in addition to hardware modules like decoders, comparators, or adders that you may need. MUXes have a 0.5 ns delay. What is the maximum frequency for your design?

An attached PowerPoint file contains templates for the blocks above. Feel free to use that for your micro-architecture diagrams.

## **Q2. Pipelined CSGY6133 [20 Points]**

Now consider a five stage pipelined implementation of the CSGY6133 processor. The five stages are described below:


- **IF:** Read the instruction from IM using the current PC. Update the PC to PC+1 for non-branch instructions. (Branch instructions are resolved in the EX stage; if the branch is taken, the PC is updated to the branch address instead of PC+1).
- **RF/ID:** Read operands for the current instruction from the RF. Decode the instruction and generate control signals.
- **Mem:** Read from or write to DM for the *subm* and *st* instructions, respectively.
- **EX:** Compute the result of *sub* and *subm* instructions by subtracting one operand from the other. (Compare the two operands for the *bleq* instruction and determine if the branch condition is true).
- **WB:** Write-back the result to the RF for the *sub* and *subm* instructions.

Two consecutive sub instructions in the CSGY6133 pipeline have a RAW dependency if the result of one is an operand of the next. For example the following two instructions have:

```
sub $1, $2
sub $2, $3
```

have a RAW dependency. The dependency can be resolved using an EX-EX forwarding path, as shown in the timing diagram below.

Cycle	1	2	3	4	5	6
sub \$1, \$2	IF	RF/ID	Mem	EX	WB	
sub \$2, \$3		IF	RF/ID	Mem	EX	WB



Consider all possible pairs of two consecutive CSGY6133 instructions (you can exclude the bleg instruction in your analysis), identify if a RAW hazard can exist for that pair and if so, draw a timing diagram such as the one above illustrating how to resolve the RAW hazard using forwarding, stalling, or both. Obviously the solution for the sub-sub pair has already been done for you. (Note that some pairs of instructions might have more than different type of RAW hazard between them.)

### **Q3. Pipelined MIPS [35 Points]**

The following MIPS assembly code executes on a 5-stage pipelined MIPS processor. The goal is to compute the sum of two arrays, A and B, and to copy the contents of the array B to array A.

```
//baseA and baseB are the base addresses of arrays A and B
//Each array has N 32-bit elements
//Reg. $1 holds loop counter i, initialized to i=-4
//Reg. $2 holds sum1, initialized to sum1=0
//Reg. $3 holds sum2, initialized to sum2=0
//Reg. $4 holds tmp1, initialized to tmp1=0
//Reg. $5 holds tmp2, initialized to tmp2=0
//Reg. $6 holds count=(N-1)*4
//Note: The operands in mnemonics below appear in the same order as in the instruction format
```

```
LOOP: addiu $1, $1, 4      //i = i+4
lw $1, $4, baseA        //tmp1 = A[i]
addi $2, $4, $2          //sum1 = tmp1 + sum1
lw $1, $5, baseB        //tmp2 = B[i]
sw $1, $5, baseA        // A[i] = tmp2
addi $3, $5, $3          //sum2 = tmp2 + sum2
bne $1, $6, LOOP        //if i != count = (N-1)*4 then goto LOOP
```

3.1 [10 Points] Assuming a standard 5-stage MIPS pipeline, identify each pair of instructions that results in a RAW hazard (including those that are non-consecutive). For each pair, draw a timing diagram illustrating how the MIPS processor resolves the dependency (using either forwarding, stalling or both). You can assume that the processor has EX-EX, MEM-EX and MEM-MEM forwarding.

3.2 [5 Points] Draw a timing diagram that illustrates the execution of the first 6 instructions of the code above (i.e., up to but not including the bneq instruction). You can assume that there are no instructions in the pipeline before the loop executes and the registers/memory are (magically) initialized.

3.3 [5 Points] Estimate the number of clock cycles the loop above would take to completely execute if  $N=10$ . Branch instructions in the MIPS pipeline are resolved in the EX stage. To address control hazards, the compiler inserts two NOPS after each branch instruction (for example, after the bne instruction in the code above).

3.4 [5 Points] Repeat problem 3.2 assuming that the MIPS pipeline does not have any forwarding. Instead, it addresses hazards by detecting RAW dependencies and stalling the pipeline till the dependency is resolved.

3.5 [10 Points] Re-order the instructions in the code above such that you achieve the same functionality, but minimize the number of cycles taken to execute the first 6 instructions. As in 3.5, assume that the MIPS processor only implements stalling.

#### **Q4. Caches [20 Points]**

4.1 [5 Points] Determine the number of offset bits, index bits and tag bits for a 1GB byte-addressable memory system has a 8 KB cache with the following parameters (separate answers for each case):

- (a) Direct-mapped, with 1 Byte blocks
- (b) Direct mapped, with 4 Byte blocks
- (c) 2-way set-associative, with 8 Byte blocks
- (d) 4-way set-associative, with 16 Byte blocks
- (e) Fully-associative, with 8 Byte blocks

4.2 [10 Points] Consider a small direct mapped 8-Byte cache with 2 Byte blocks. The cache is accessed with the following sequence of addresses. (The cache is Byte addressable so each access returns a Byte of data.)

0,1,2,3,4,5,6,7,8,9, 0,1,2,3,4,5,6,7,8,9 ...

Determine if each access is a Hit or a Miss. If Miss, determine if the miss is a compulsory, conflict or capacity miss. You can assume the cache is initially cold/empty.

4.3 [5 Points] Now assume that we add a small 2-entry victim cache (each entry hold an entire cache block) to the direct mapped cache above. For the same pattern of accesses, determine which ones hit in the cache and which ones miss. On a hit, indicate if the hit was in the main cache or the victim cache.